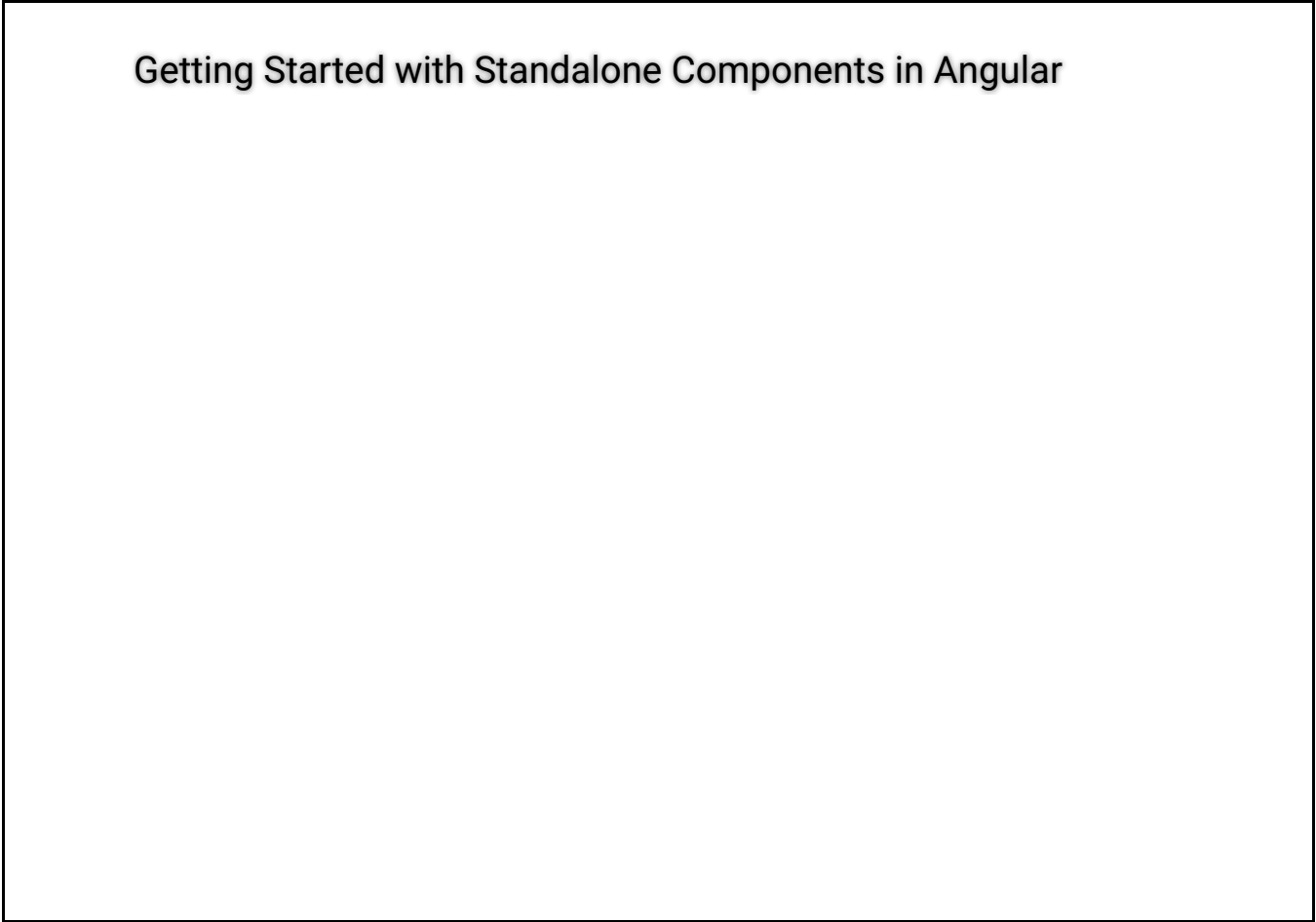


# Getting started with standalone components

**Standalone components** provide a simplified way to build Angular applications. Standalone components, directives, and pipes aim to streamline the authoring experience by reducing the need for [NgModules](#). Existing applications can optionally and incrementally adopt the new standalone style without any breaking changes.

## Creating standalone components



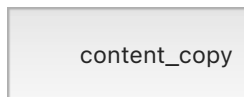
Getting Started with Standalone Components in Angular

## The standalone flag and component imports

Components, directives, and pipes can now be marked as `standalone: true`. Angular classes marked as standalone do not need to be declared in

an [NgModule](#) (the Angular compiler will report an error if you try).

Standalone components specify their dependencies directly instead of getting them through [NgModules](#). For example, if `PhotoGalleryComponent` is a standalone component, it can directly import another standalone component `ImageGridComponent`:



```
@Component({
  standalone: true,
  selector: 'photo-gallery',
  imports: [ImageGridComponent],
  template: `
    ... <image-grid [images]="imageList"></image-grid>
  `,
})
export class PhotoGalleryComponent {
  // component logic
}
```

`imports` can also be used to reference standalone directives and pipes. In this way, standalone components can be written without the need to create an [NgModule](#) to manage template dependencies.

## Using existing NgModules in a standalone component

When writing a standalone component, you may want to use other components, directives, or pipes in the component's template. Some of

those dependencies might not be marked as standalone, but instead declared and exported by an existing [NgModule](#). In this case, you can import the [NgModule](#) directly into the standalone component:

content\_copy

```
@Component({
  standalone: true,
  selector: 'photo-gallery',
  // an existing module is imported directly into a standalone component
  imports: [MatButtonModule],
  template: `
    ...
    <button mat-button>Next Page</button>
  `,
})
export class PhotoGalleryComponent {
  // logic
}
```

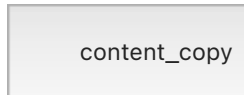
You can use standalone components with existing [NgModule](#)-based libraries or dependencies in your template. Standalone components can take full advantage of the existing ecosystem of Angular libraries.

## Using standalone components in NgModule-based applications

Standalone components can also be imported into existing NgModules-based contexts. This allows existing applications (which are using

NgModules today) to incrementally adopt the new, standalone style of component.

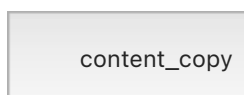
You can import a standalone component (or directive, or pipe) just like you would an [NgModule](#) - using [NgModule.imports](#):



```
@NgModule({  
  declarations: [AlbumComponent],  
  exports: [AlbumComponent],  
  imports: [PhotoGalleryComponent],  
})  
export class AlbumModule {}
```

## Bootstrapping an application using a standalone component

An Angular application can be bootstrapped without any [NgModule](#) by using a standalone component as the application's root component. This is done using the [bootstrapApplication](#) API:



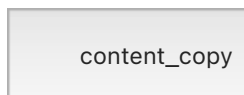
```
// in the main.ts file  
import {bootstrapApplication} from '@angular/platform-browser';
```

```
import {PhotoAppComponent} from './app/photo.app.component';  
  
bootstrapApplication(PhotoAppComponent);
```

## Configuring dependency injection

When bootstrapping an application, often you want to configure Angular's dependency injection and provide configuration values or services for use throughout the application. You can pass these as providers to

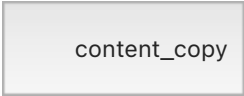
[bootstrapApplication](#):



```
bootstrapApplication(PhotoAppComponent, {  
  providers: [  
    {provide: BACKEND_URL, useValue: 'https://photoapp.looknongmodules.com/api'},  
    // ...  
  ]  
});
```

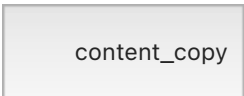
The standalone bootstrap operation is based on explicitly configuring a list of [Providers](#) for dependency injection. In Angular, provide-prefixed functions can be used to configure different systems without needing to

import NgModules. For example, [provideRouter](#) is used in place of RouterModule.forRoot to configure the router:

A small rectangular button with a light gray background and a thin black border. Inside the button, the text "content\_copy" is written in a dark gray, sans-serif font.

```
bootstrapApplication(PhotoAppComponent, {  
  providers: [  
    {provide: BACKEND_URL, useValue: 'https://photoapp.looknongmodules.com/api'},  
    provideRouter([/* app routes */]),  
    // ...  
  ]  
});
```

Many third party libraries have also been updated to support this provide-function configuration pattern. If a library only offers an NgModule API for its DI configuration, you can use the [importProvidersFrom](#) utility to still use it with [bootstrapApplication](#) and other standalone contexts:

A small rectangular button with a light gray background and a thin black border. Inside the button, the text "content\_copy" is written in a dark gray, sans-serif font.

```
import {LibraryModule} from 'ngmodule-based-library';  
  
bootstrapApplication(PhotoAppComponent, {  
  providers: [  
    {provide: BACKEND_URL, useValue: 'https://photoapp.looknongmodules.com/api'},  
    importProvidersFrom(  
      LibraryModule.forRoot()  
    )  
  ]  
});
```

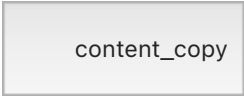
```
    ),  
  ],  
});
```

## Routing and lazy-loading

The router APIs were updated and simplified to take advantage of the standalone components: an [NgModule](#) is no longer required in many common, lazy-loading scenarios.

### Lazy loading a standalone component

Any route can lazily load its routed, standalone component by using `loadComponent`:



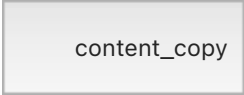
content\_copy

```
export const ROUTES: Route[] = [  
  {path: 'admin', loadComponent: () => import('./admin/panel.component').then(mod =>  
mod.AdminPanelComponent)},  
  // ...  
];
```

This works as long as the loaded component is standalone.

## Lazy loading many routes at once

The `loadChildren` operation now supports loading a new set of child [Routes](#) without needing to write a lazy loaded [NgModule](#) that imports `RouterModule.forChild` to declare the routes. This works when every route loaded this way is using a standalone component.

content\_copy

```
// In the main application:
export const ROUTES: Route[] = [
  {path: 'admin', loadChildren: () => import('./admin/routes').then(mod =>
mod.ADMIN_ROUTES)},
  // ...
];

// In admin/routes.ts:
export const ADMIN_ROUTES: Route[] = [
  {path: 'home', component: AdminHomeComponent},
  {path: 'users', component: AdminUsersComponent},
  // ...
];
```

## Lazy loading and default exports

When using `loadChildren` and `loadComponent`, the router understands and automatically unwraps dynamic `import()` calls with default exports. You can take advantage of this to skip the `.then()` for such lazy loading operations.



content\_copy

```
// In the main application:
export const ROUTES: Route[] = [
  {path: 'admin', loadChildren: () => import('./admin/routes')},
  // ...
];

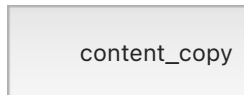
// In admin/routes.ts:
export default [
  {path: 'home', component: AdminHomeComponent},
  {path: 'users', component: AdminUsersComponent},
  // ...
] as Route[];
```

## Providing services to a subset of routes

The lazy loading API for [NgModules](#) (`loadChildren`) creates a new "module" injector when it loads the lazily loaded children of a route. This feature was often useful to provide services only to a subset of routes in the application. For example, if all routes under `/admin` were scoped using a `loadChildren` boundary, then admin-only services could be provided only to those routes. Doing this required using the `loadChildren` API, even if lazy loading of the routes in question was unnecessary.

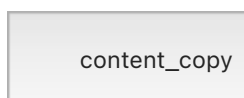
The Router now supports explicitly specifying additional providers on a [Route](#), which allows this same scoping without the need for either lazy loading or [NgModules](#). For example, scoped services within an `/admin`

route structure would look like:



```
export const ROUTES: Route[] = [
{
  path: 'admin',
  providers: [
    AdminService,
    {provide: ADMIN_API_KEY, useValue: '12345'},
  ],
  children: [
    {path: 'users', component: AdminUsersComponent},
    {path: 'teams', component: AdminTeamsComponent},
  ],
},
// ... other application routes that don't
// have access to ADMIN_API_KEY or AdminService.
];
```

It's also possible to combine providers with `loadChildren` of additional routing configuration, to achieve the same effect of lazy loading an [NgModule](#) with additional routes and route-level providers. This example configures the same providers/child routes as above, but behind a lazy loaded boundary:



```
// Main application:
export const ROUTES: Route[] = {
  // Lazy-load the admin routes.
  {path: 'admin', loadChildren: () => import('./admin/routes').then(mod =>
mod.ADMIN_ROUTES)},
  // ... rest of the routes
}

// In admin/routes.ts:
export const ADMIN_ROUTES: Route[] = [{
  path: '',
  pathMatch: 'prefix',
  providers: [
    AdminService,
    {provide: ADMIN_API_KEY, useValue: 12345},
  ],
  children: [
    {path: 'users', component: AdminUsersCmp},
    {path: 'teams', component: AdminTeamsCmp},
  ],
}];
```

Note the use of an empty-path route to host providers that are shared among all the child routes.

[importProvidersFrom](#) can be used to import existing NgModule-based DI configuration into route providers as well.

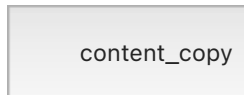
## Advanced topics

This section goes into more details that are relevant only to more advanced usage patterns. You can safely skip this section when learning

about standalone components, directives, and pipes for the first time.

## Standalone components for library authors

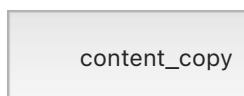
Standalone components, directives, and pipes can be exported from [NgModules](#) that import them:



```
@NgModule({  
  imports: [ImageCarouselComponent, ImageSlideComponent],  
  exports: [ImageCarouselComponent, ImageSlideComponent],  
})  
export class CarouselModule {}
```

This pattern is useful for Angular libraries that publish a set of cooperating directives. In the above example, both the `ImageCarouselComponent` and `ImageSlideComponent` need to be present in a template to build up one logical "carousel widget".

As an alternative to publishing a [NgModule](#), library authors might want to export an array of cooperating directives:



```
export const CAROUSEL_DIRECTIVES = [ImageCarouselComponent,  
ImageSlideComponent] as const;
```

Such an array could be imported by applications using [NgModules](#) and added to the [@NgModule.imports](#). Please note the presence of the TypeScript's `as const` construct: it gives Angular compiler additional information required for proper compilation and is a recommended practice (as it makes the exported array immutable from the TypeScript point of view).

## Dependency injection and injectors hierarchy

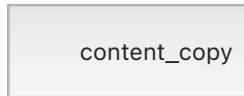
Angular applications can configure dependency injection by specifying a set of available providers. In a typical application, there are two different injector types:

- **module injector** with providers configured in [@NgModule.providers](#) or [@Injectable\({providedIn: "..."}\)](#). Those application-wide providers are visible to all components in as well as to other services configured in a module injector.
- **node injectors** configured in [@Directive.providers](#) / [@Component.providers](#) or [@Component.viewProviders](#). Those providers are visible to a given component and all its children only.

### Environment injectors

Making [NgModules](#) optional will require new ways of configuring "module" injectors with application-wide providers (for example, [HttpClient](#)). In the standalone application (one created with [bootstrapApplication](#)),

“module” providers can be configured during the bootstrap process, in the providers option:



```
bootstrapApplication(PhotoAppComponent, {  
  providers: [  
    {provide: BACKEND_URL, useValue: 'https://photoapp.looknongmodules.com/api'},  
    {provide: PhotosService, useClass: PhotosService},  
    // ...  
  ]  
});
```

The new bootstrap API gives us back the means of configuring “module injectors” without using [NgModules](#). In this sense, the “module” part of the name is no longer relevant and we’ve decided to introduce a new term: “environment injectors”.

Environment injectors can be configured using one of the following:

- [@NgModule.providers](#) (in applications bootstrapping through an [NgModule](#));
- [@Injectable](#)({provideIn: "..."}) (in both the NgModule-based and the “standalone” applications);
- providers option in the [bootstrapApplication](#) call (in fully “standalone” applications);
- providers field in a [Route](#) configuration.

Angular v14 introduces a new TypeScript type [EnvironmentInjector](#) to represent this new naming. The accompanying [createEnvironmentInjector](#) API makes it possible to create environment injectors programmatically:

content\_copy

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

```
const parentInjector = ... // existing environment injector
const childInjector = createEnvironmentInjector([{provide: PhotosService, useClass:
CustomPhotosService}], parentInjector);
```

Environment injectors have one additional capability: they can execute initialization logic when an environment injector gets created (similar to the [NgModule](#) constructors that get executed when a module injector is created):

content\_copy

```
import {createEnvironmentInjector, ENVIRONMENT\_INITIALIZER} from
 '@angular/core';

createEnvironmentInjector([
  {provide: PhotosService, useClass: CustomPhotosService},
  {provide: ENVIRONMENT\_INITIALIZER, useValue: () => {
```

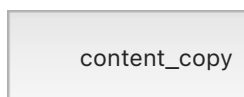
```

        console.log("This function runs when this EnvironmentInjector gets created");
    }}
    });

```

## Standalone injectors

In reality, the dependency injectors hierarchy is slightly more elaborate in applications using standalone components. Let's consider the following example:



```

// an existing "datepicker" component with an NgModule
@Component({
    selector: 'datepicker',
    template: '...',
})
class DatePickerComponent {
    constructor(private calendar: CalendarService) {}
}

@NgModule({
    declarations: [DatePickerComponent],
    exports: [DatePickerComponent]
    providers: [CalendarService],
})
class DatePickerModule {}

@Component({
    selector: 'date-modal',
    template: '<datepicker></datepicker>',

```



```
    standalone: true,  
    imports: [DatePickerModule]  
  })  
  class DateModalComponent {  
  }
```

In the above example, the component `DateModalComponent` is standalone - it can be consumed directly and has no `NgModule` which needs to be imported in order to use it. However, `DateModalComponent` has a dependency, the `DatePickerComponent`, which is imported via its `NgModule` (the `DatePickerModule`). This `NgModule` may declare providers (in this case: `CalendarService`) which are required for the `DatePickerComponent` to function correctly.

When Angular creates a standalone component, it needs to know that the current injector has all of the necessary services for the standalone component's dependencies, including those based on `NgModules`. To guarantee that, in some cases Angular will create a new "standalone injector" as a child of the current environment injector. Today, this happens for all bootstrapped standalone components: it will be a child of the root environment injector. The same rule applies to the dynamically created (for example, by the router or the [ViewContainerRef](#) API) standalone components.

A separate standalone injector is created to ensure that providers imported by a standalone component are "isolated" from the rest of the application. This lets us think of standalone components as truly self-contained pieces that can't "leak" their implementation details to the rest of the application.