# <https://formden.com/form-builder/>

# Modules

 An NgModule declares a compilation context for a set of components that is dedicated to an application domain, a workflow, or a closely related set of capabilities. An NgModule can associate its components with related code, such as services, to form functional units.

Every Angular app has a root module, conventionally named AppModule, which provides the bootstrap mechanism that launches the application. An app typically contains many functional modules.

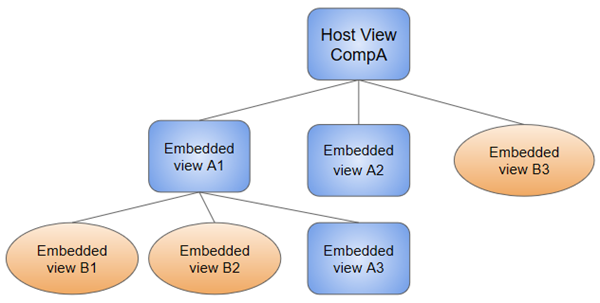
Like JavaScript modules, NgModules can import functionality from other NgModules, and allow their own functionality to be exported and used by other NgModules. For example, to use the router service in your app, you import the [Router](https://angular.io/api/router/Router) NgModule.

An NgModule is defined by a class decorated with @[NgModule](https://angular.io/api/core/NgModule)(). The @[NgModule](https://angular.io/api/core/NgModule)() decorator is a function that takes a single metadata object, whose properties describe the module. The most important properties are as follows.

* declarations: The [components](https://angular.io/guide/architecture-components), directives, and pipes that belong to this NgModule.
* exports: The subset of declarations that should be visible and usable in the component templates of other NgModules.
* imports: Other modules whose exported classes are needed by component templates declared in this NgModule.
* providers: Creators of [services](https://angular.io/guide/architecture-services) that this NgModule contributes to the global collection of services; they become accessible in all parts of the app. (You can also specify providers at the component level, which is often preferred.)
* bootstrap: The main application view, called the root component, which hosts all other app views. Only the root NgModule should set the bootstrap property.

A root NgModule always has a root component that is created during bootstrap, but any NgModule can include any number of additional components, which can be loaded through the router or created through the template.

A component and its template together define a *view*. A component can contain a *view hierarchy*, which allows you to define arbitrarily complex areas of the screen that can be created, modified, and destroyed as a unit. A view hierarchy can mix views defined in components that belong to different NgModules. This is often the case, especially for UI libraries.



When you create a component, it's associated directly with a single view, called the *host view*. The host view can be the root of a view hierarchy, which can contain *embedded views*, which are in turn the host views of other components. Those components can be in the same NgModule, or can be imported from other NgModules. Views in the tree can be nested to any depth.

Angular loads as a collection of JavaScript modules. You can think of them as library modules. Each Angular library name begins with the @angular prefix. Install them with the node package manager npm and import parts of them with JavaScript import statements.

For example, import Angular's [Component](https://angular.io/api/core/Component) decorator from the @angular/core library like this.

import { [Component](https://angular.io/api/core/Component) } from '@angular/core';

# Components

Every Angular application has at least one component, the root component that connects a component hierarchy with the page document object model (DOM). Each component defines a class that contains application data and logic, and is associated with an HTML template that defines a view to be displayed in a target environment.

The @[Component](https://angular.io/api/core/Component)() decorator identifies the class immediately below it as a component, and provides the template and related component-specific metadata.

The metadata for a component tells Angular where to get the major building blocks that it needs to create and present the component and its view. In particular, it associates a template with the component, either directly with inline code, or by reference. Together, the component and its template describe a view.

Here's an example of basic metadata for HeroListComponent.

src/app/hero-list.component.ts (metadata)

@[Component](https://angular.io/api/core/Component)({

selector: 'app-hero-list',

templateUrl: './hero-list.component.html',

providers: [ HeroService ]

})

export class HeroListComponent implements [OnInit](https://angular.io/api/core/OnInit) {

/\* . . . \*/

}

This example shows some of the most useful @[Component](https://angular.io/api/core/Component) configuration options:

* selector: A CSS selector that tells Angular to create and insert an instance of this component wherever it finds the corresponding tag in template HTML. For example, if an app's HTML contains <app-hero-list></app-hero-list>, then Angular inserts an instance of the HeroListComponent view between those tags.
* templateUrl: The module-relative address of this component's HTML template. Alternatively, you can provide the HTML template inline, as the value of the template property. This template defines the component's host view.
* providers: An array of [providers](https://angular.io/guide/glossary#provider) for services that the component requires. In the example, this tells Angular how to provide the HeroService instance that the component's constructor uses to get the list of heroes to display.

### **Data binding**

Without a framework, you would be responsible for pushing data values into the HTML controls and turning user responses into actions and value updates. Writing such push and pull logic by hand is tedious, error-prone, and a nightmare to read, as any experienced front-end JavaScript programmer can attest.

Angular supports two-way data binding, a mechanism for coordinating the parts of a template with the parts of a component. Add binding markup to the template HTML to tell Angular how to connect both sides.

The following diagram shows the four forms of data binding markup. Each form has a direction: to the DOM, from the DOM, or both.



This example from the HeroListComponent template uses three of these forms.

src/app/hero-list.component.html (binding)

<li>{{hero.name}}</li>

<app-hero-detail [hero]="selectedHero"></app-hero-detail>

<li (click)="selectHero(hero)"></li>

* The {{hero.name}} [*interpolation*](https://angular.io/guide/displaying-data#interpolation) displays the component's hero.name property value within the <li> element.
* The [hero] [*property binding*](https://angular.io/guide/property-binding) passes the value of selectedHero from the parent HeroListComponent to the hero property of the child HeroDetailComponent.
* The (click) [*event binding*](https://angular.io/guide/user-input#binding-to-user-input-events) calls the component's selectHero method when the user clicks a hero's name.

Two-way data binding (used mainly in [template-driven forms](https://angular.io/guide/forms)) combines property and event binding in a single notation. Here's an example from the HeroDetailComponent template that uses two-way data binding with the [ngModel](https://angular.io/api/forms/NgModel) directive.

src/app/hero-detail.component.html (ngModel)

<input [([ngModel](https://angular.io/api/forms/NgModel))]="hero.name">

### **Directives**

Angular templates are dynamic. When Angular renders them, it transforms the DOM according to the instructions given by directives. A directive is a class with a @[Directive](https://angular.io/api/core/Directive)() decorator.

A component is technically a directive. However, components are so distinctive and central to Angular applications that Angular defines the @[Component](https://angular.io/api/core/Component)() decorator, which extends the @[Directive](https://angular.io/api/core/Directive)() decorator with template-oriented features.

In addition to components, there are two other kinds of directives: structural and attribute. Angular defines a number of directives of both kinds, and you can define your own using the @[Directive](https://angular.io/api/core/Directive)() decorator.

Just as for components, the metadata for a directive associates the decorated class with a selector element that you use to insert it into HTML. In templates, directives typically appear within an element tag as attributes, either by name or as the target of an assignment or a binding.

#### **Structural directives**

Structural directives alter layout by adding, removing, and replacing elements in the DOM. The example template uses two built-in structural directives to add application logic to how the view is rendered.

src/app/hero-list.component.html (structural)

<li \*[ngFor](https://angular.io/api/common/NgForOf)="let hero of heroes"></li>

<app-hero-detail \*[ngIf](https://angular.io/api/common/NgIf)="selectedHero"></app-hero-detail>

* [\*ngFor](https://angular.io/guide/displaying-data#ngFor) is an iterative; it tells Angular to stamp out one <li> per hero in the heroes list.
* [\*ngIf](https://angular.io/guide/displaying-data#ngIf) is a conditional; it includes the HeroDetail component only if a selected hero exists.

#### **Attribute directives**

Attribute directives alter the appearance or behavior of an existing element. In templates they look like regular HTML attributes, hence the name.

The [ngModel](https://angular.io/api/forms/NgModel) directive, which implements two-way data binding, is an example of an attribute directive. [ngModel](https://angular.io/api/forms/NgModel) modifies the behavior of an existing element (typically <input>) by setting its display value property and responding to change events.

src/app/hero-detail.component.html (ngModel)

<input [([ngModel](https://angular.io/api/forms/NgModel))]="hero.name">

# Services and dependency injection

For data or logic that isn't associated with a specific view, and that you want to share across components, you create a service class. A service class definition is immediately preceded by the @[Injectable](https://angular.io/api/core/Injectable)() decorator. The decorator provides the metadata that allows other providers to be **injected** as dependencies into your class.

DI is wired into the Angular framework and used everywhere to provide new components with the services or other things they need. Components consume services; that is, you can inject a service into a component, giving the component access to that service class.

To define a class as a service in Angular, use the @[Injectable](https://angular.io/api/core/Injectable)() decorator to provide the metadata that allows Angular to inject it into a component as a dependency. Similarly, use the @[Injectable](https://angular.io/api/core/Injectable)() decorator to indicate that a component or other class (such as another service, a pipe, or an NgModule) has a dependency.

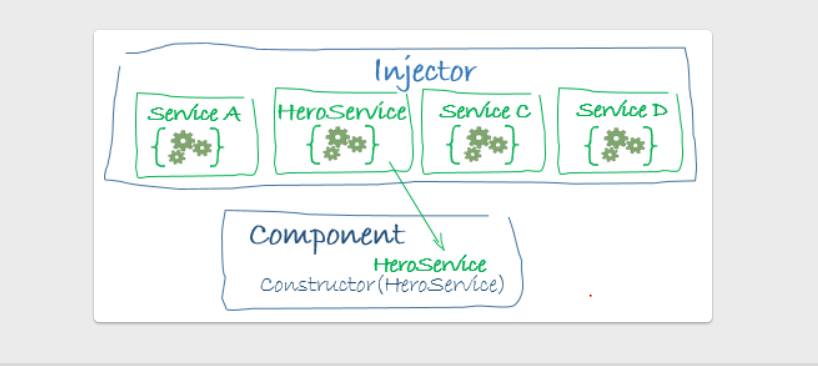
When Angular creates a new instance of a component class, it determines which services or other dependencies that component needs by looking at the constructor parameter types. For example, the constructor of HeroListComponent needs HeroService.

src/app/hero-list.component.ts (constructor)

constructor(private service: HeroService) { }

When Angular discovers that a component depends on a service, it first checks if the injector has any existing instances of that service. If a requested service instance doesn't yet exist, the injector makes one using the registered provider, and adds it to the injector before returning the service to Angular.

When all requested services have been resolved and returned, Angular can call the component's constructor with those services as arguments.



You must register at least one *provider* of any service you are going to use. The provider can be part of the service's own metadata, making that service available everywhere, or you can register providers with specific modules or components. You register providers in the metadata of the service (in the @[Injectable](https://angular.io/api/core/Injectable)() decorator), or in the @[NgModule](https://angular.io/api/core/NgModule)() or @[Component](https://angular.io/api/core/Component)() metadata

* By default, the Angular CLI command [ng generate service](https://angular.io/cli/generate) registers a provider with the root injector for your service by including provider metadata in the @[Injectable](https://angular.io/api/core/Injectable)() decorator. The tutorial uses this method to register the provider of HeroService class definition.

@[Injectable](https://angular.io/api/core/Injectable)({

providedIn: 'root',

})

When you provide the service at the root level, Angular creates a single, shared instance of HeroService and injects it into any class that asks for it. Registering the provider in the @[Injectable](https://angular.io/api/core/Injectable)() metadata also allows Angular to optimize an app by removing the service from the compiled app if it isn't used.

When you register a provider with a [specific NgModule](https://angular.io/guide/architecture-modules), the same instance of a service is available to all components in that NgModule. To register at this level, use the providers property of the @[NgModule](https://angular.io/api/core/NgModule)() decorator,

@[NgModule](https://angular.io/api/core/NgModule)({

providers: [

BackendService,

Logger

],

...

})

When you register a provider at the component level, you get a new instance of the service with each new instance of that component. At the component level, register a service provider in the providers property of the @[Component](https://angular.io/api/core/Component)() metadata.

src/app/hero-list.component.ts (component providers)

@[Component](https://angular.io/api/core/Component)({

selector: 'app-hero-list',

templateUrl: './hero-list.component.html',

providers: [ HeroService ]

})

# template-driven form

<https://angular.io/guide/forms>

# Intercetor

<https://medium.com/@ryanchenkie_40935/angular-authentication-using-the-http-client-and-http-interceptors-2f9d1540eb8>

<https://jasonwatmore.com/post/2019/06/10/angular-8-user-registration-and-login-example-tutorial>

<https://medium.com/@mool.smreeti/microservices-with-spring-boot-authentication-with-jwt-and-spring-security-6e10155d9db0>

# **Observables in Angular**

Angular makes use of observables as an interface to handle a variety of common asynchronous operations. For example:

* You can define [custom events](https://angular.io/guide/event-binding#custom-events-with-eventemitter) that send observable output data from a child to a parent component.
* The HTTP module uses observables to handle AJAX requests and responses.
* The Router and Forms modules use observables to listen for and respond to user-input events.

Create observable source in service

private emitChangeSource = new Subject<any>();

changeEmitted$ = this.emitChangeSource.asObservable();

emit data when below method is called with data

emitChange(data: {}) {

    this.emitChangeSource.next(data);

  }

Subscribe the emited data in other component or service

sharedService.changeEmitted$.subscribe(data => {

      this.isLoading = data.isLoading;

    })

# Angular Promises Versus Observables

<https://www.syncfusion.com/blogs/post/angular-promises-versus-observables.aspx>

# RxJS 6 Sources: Map and Pipe

<https://medium.com/angular-in-depth/reading-the-rxjs-6-sources-map-and-pipe-94d51fec71c2>

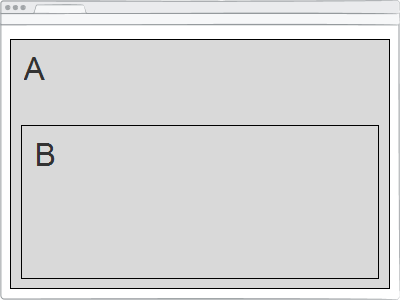
# How to pass data between routed components in Angular

There are several ways how Angular components can pass data around:

* Using @Input and @Output
* By injecting parent component through constructor or child components through @ViewChild, @ViewChildren, @ContentChild, @ContentChildren and directly calling component’s API
* Using services (this covers state management libraries like [ngrx](https://ngrx.io/))
* Using router parameters

First two methods can only be used when we have parent — child relationship between our components — one of the components is the parent node in the DOM:

Image for post



When we have routed components, then it looks like this:

Image for post



There is no parent child relationship between components A and B, so we cannot use first two methods to pass the data. We could use route parameters and pass some identifier to the routed component and the data could be resolved through router but imagine that we don’t have an id and we want to pass rather big object to other component. We also don’t want to have all the object properties in query parameters. How can Component A pass some data to Component B? We can solve it by using service: