1. **Architecture overview**

Angular is a platform and framework for building client applications in HTML and TypeScript. Angular is written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you import into your apps.

The basic building blocks of an Angular application are *NgModules*, which provide a compilation context for *components*. NgModules collect related code into functional sets; an Angular app is defined by a set of NgModules. An app always has at least a *root module* that enables bootstrapping, and typically has many more *feature modules*.

* Components define *views*, which are sets of screen elements that Angular can choose among and modify according to your program logic and data.
* Components use *services*, which provide specific functionality not directly related to views. Service providers can be *injected* into components as *dependencies*, making your code modular, reusable, and efficient.

Both components and services are simply classes, with *decorators* that mark their type and provide metadata that tells Angular how to use them.

* The metadata for a component class associates it with a *template* that defines a view. A template combines ordinary HTML with Angular *directives* and *binding markup* that allow Angular to modify the HTML before rendering it for display.
* The metadata for a service class provides the information Angular needs to make it available to components through *dependency injection (DI)*.

An app's components typically define many views, arranged hierarchically. Angular provides the [Router](https://angular.io/api/router/Router) service to help you define navigation paths among views. The router provides sophisticated in-browser navigational capabilities.

## **Modules**

Angular NgModules differ from and complement JavaScript (ES2015) 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.

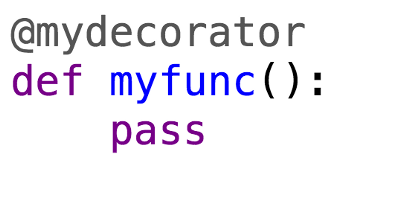
Every Angular app has a root module, conventionally named AppModule, which provides the bootstrap mechanism that launches the application. An app typically contains many functional modules.

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.

Organizing your code into distinct functional modules helps in managing development of complex applications, and in designing for reusability. In addition, this technique lets you take advantage of lazy-loading—that is, loading modules on demand—to minimize the amount of code that needs to be loaded at startup.

### The Decorator Pattern

What the heck are decorators anyway? Well, in Python, decorators provide a very simple syntax for calling [higher-order](https://en.wikipedia.org/wiki/Higher-order_function) functions. A Python decorator is a function that takes another function, extending the behavior of the latter function without explicitly modifying it. The [simplest](http://www.saltycrane.com/blog/2010/03/simple-python-decorator-examples/) decorator in Python could look like this:



That thing at the very top (`@mydecorator`) is a decorator and isn’t going to look that different in ES2016 (ES7), so pay attention! :).

*`@`* indicates to the parser that we’re using a decorator while *mydecorator* references a function by that name. Our decorator takes an argument (the function being decorated) and returns the same function with added functionality.

Decorators are helpful for anything you want to transparently wrap with extra functionality. These include memoization, enforcing access control and authentication, instrumentation and timing functions, logging, rate-limiting, and the list goes on.

## **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.

Decorators are functions that modify JavaScript classes. Angular defines a number of decorators that attach specific kinds of metadata to classes, so that the system knows what those classes mean and how they should work.

### **Templates, directives, and data binding**

A template combines HTML with Angular markup that can modify HTML elements before they are displayed. Template directives provide program logic, and binding markup connects your application data and the DOM. There are two types of data binding:

* Event binding lets your app respond to user input in the target environment by updating your application data.
* Property binding lets you interpolate values that are computed from your application data into the HTML.

Before a view is displayed, Angular evaluates the directives and resolves the binding syntax in the template to modify the HTML elements and the DOM, according to your program data and logic. Angular supports two-way data binding, meaning that changes in the DOM, such as user choices, are also reflected in your program data.

Your templates can use pipes to improve the user experience by transforming values for display. For example, use pipes to display dates and currency values that are appropriate for a user's locale. Angular provides predefined pipes for common transformations, and you can also define your own pipes.

## **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 your service to be injected into client components as a dependency.

Dependency injection (DI) lets you keep your component classes lean and efficient. They don't fetch data from the server, validate user input, or log directly to the console; they delegate such tasks to services.

### **Routing**

The Angular [Router](https://angular.io/api/router/Router) NgModule provides a service that lets you define a navigation path among the different application states and view hierarchies in your app. It is modeled on the familiar browser navigation conventions:

* Enter a URL in the address bar and the browser navigates to a corresponding page.
* Click links on the page and the browser navigates to a new page.
* Click the browser's back and forward buttons and the browser navigates backward and forward through the history of pages you've seen.

The router maps URL-like paths to views instead of pages. When a user performs an action, such as clicking a link, that would load a new page in the browser, the router intercepts the browser's behavior, and shows or hides view hierarchies.

If the router determines that the current application state requires particular functionality, and the module that defines it hasn't been loaded, the router can lazy-load the module on demand.

The router interprets a link URL according to your app's view navigation rules and data state. You can navigate to new views when the user clicks a button or selects from a drop box, or in response to some other stimulus from any source. The router logs activity in the browser's history, so the back and forward buttons work as well.

To define navigation rules, you associate navigation paths with your components. A path uses a URL-like syntax that integrates your program data, in much the same way that template syntax integrates your views with your program data. You can then apply program logic to choose which views to show or to hide, in response to user input and your own access rules.

## **What's next**

You've learned the basics about the main building blocks of an Angular application. The following diagram shows how these basic pieces are related.



* Together, a component and template define an Angular view.
  + A decorator on a component class adds the metadata, including a pointer to the associated template.
  + Directives and binding markup in a component's template modify views based on program data and logic.
* The dependency injector provides services to a component, such as the router service that lets you define navigation among views.

### **Modules in JavaScript**

Even though JavaScript never had built-in modules, the community has converged on a simple style of modules, which is supported by libraries in ES5 and earlier. This style has also been adopted by ES6:

* Each module is a piece of code that is executed once it is loaded.
* In that code, there may be declarations (variable declarations, function declarations, etc.).
  + By default, these declarations stay local to the module.
  + You can mark some of them as exports, then other modules can import them.
* A module can import things from other modules. It refers to those modules via module specifiers, strings that are either:
  + Relative paths ('../model/user'): these paths are interpreted relatively to the location of the importing module. The file extension .js can usually be omitted.
  + Absolute paths ('/lib/js/helpers'): point directly to the file of the module to be imported.
  + Names ('util'): What modules names refer to has to be configured.
* Modules are singletons. Even if a module is imported multiple times, only a single “instance” of it exists.

This approach to modules avoids global variables, the only things that are global are module specifiers.

# Introduction to modules

Angular apps are modular and Angular has its own modularity system called NgModules. NgModules are containers for a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities. They can contain components, service providers, and other code files whose scope is defined by the containing NgModule. They can import functionality that is exported from other NgModules, and export selected functionality for use by other NgModules.

Every Angular app has at least one NgModule class, [the root module](https://angular.io/guide/bootstrapping), which is conventionally named AppModule and resides in a file named app.module.ts. You launch your app by bootstrapping the root NgModule.

While a small application might have only one NgModule, most apps have many more feature modules. The root NgModule for an app is so named because it can include child NgModules in a hierarchy of any depth.

## **NgModule metadata**

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](https://angular.io/api/core/NgModule#declarations): The [components](https://angular.io/guide/architecture-components), directives, and pipes that belong to this NgModule.
* [exports](https://angular.io/api/core/NgModule#exports): The subset of declarations that should be visible and usable in the component templates of other NgModules.
* [imports](https://angular.io/api/core/NgModule#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](https://angular.io/api/core/NgModule#bootstrap): The main application view, called the root component, which hosts all other app views. Only the root NgModule should set the [bootstrap](https://angular.io/api/core/NgModule#bootstrap) property.

Here's a simple root NgModule definition.

src/app/app.module.ts

content\_copyimport { [NgModule](https://angular.io/api/core/NgModule) } from '@angular/core';

import { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';

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

[imports](https://angular.io/api/core/NgModule#imports): [ [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) ],

providers: [ Logger ],

[declarations](https://angular.io/api/core/NgModule#declarations): [ AppComponent ],

[exports](https://angular.io/api/core/NgModule#exports): [ AppComponent ],

[bootstrap](https://angular.io/api/core/NgModule#bootstrap): [ AppComponent ]

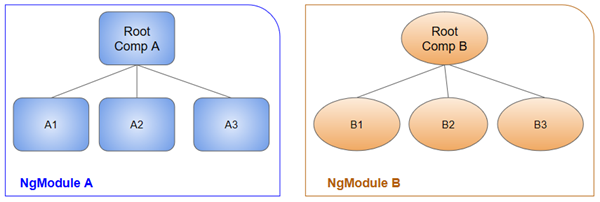
})

export class AppModule { }

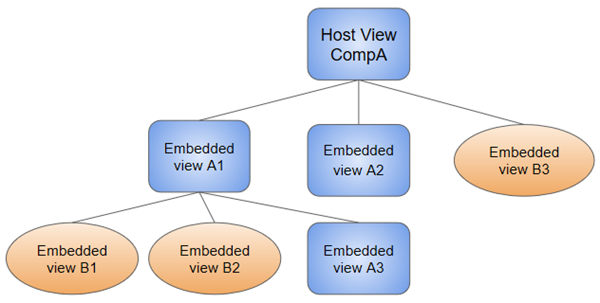
The export property of AppComponent is included here for illustration; it isn't actually necessary in this example. A root NgModule has no reason to export anything because other modules don't need to import the root NgModule.

## **NgModules and components**

NgModules provide a compilation context for their components. 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. The components that belong to an NgModule share a compilation context.

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.

\*\*Note:\*\* The hierarchical structure of views is a key factor in the way Angular detects and responds to changes in the DOM and app data.

## **NgModules and JavaScript modules**

The NgModule system is different from and unrelated to the JavaScript (ES2015) module system for managing collections of JavaScript objects. These are complementary module systems that you can use together to write your apps.

In JavaScript each file is a module and all objects defined in the file belong to that module. The module declares some objects to be public by marking them with the export key word. Other JavaScript modules use import statements to access public objects from other modules.

content\_copyimport { [NgModule](https://angular.io/api/core/NgModule) } from '@angular/core';

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

content\_copyexport class AppModule { }

[Learn more about the JavaScript module system on the web.](http://exploringjs.com/es6/ch_modules.html)

## **Angular libraries**



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 npm package manager 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.

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

You also import NgModules from Angular libraries using JavaScript import statements. For example, the following code imports the [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) NgModule from the platform-browserlibrary.

content\_copyimport { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';

In the example of the simple root module above, the application module needs material from within [BrowserModule](https://angular.io/api/platform-browser/BrowserModule). To access that material, add it to the @[NgModule](https://angular.io/api/core/NgModule) metadata [imports](https://angular.io/api/core/NgModule#imports) like this.

content\_copy[imports](https://angular.io/api/core/NgModule#imports): [ [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) ],

In this way you're using the Angular and JavaScript module systems together. Although it's easy to confuse the two systems, which share the common vocabulary of "imports" and "exports", you will become familiar with the different contexts in which they are used.

# Introduction to components

A component controls a patch of screen called a view. For example, individual components define and control each of the following views from the [Tutorial](https://angular.io/tutorial):

* The app root with the navigation links.
* The list of heroes.
* The hero editor.

You define a component's application logic—what it does to support the view—inside a class. The class interacts with the view through an API of properties and methods.

For example, HeroListComponent has a heroes property that holds an array of heroes. Its selectHero() method sets a selectedHero property when the user clicks to choose a hero from that list. The component acquires the heroes from a service, which is a TypeScript [parameter property](http://www.typescriptlang.org/docs/handbook/classes.html#parameter-properties) on the constructor. The service is provided to the component through the dependency injection system.

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

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

heroes: Hero[];

selectedHero: Hero;

constructor(private service: HeroService) { }

ngOnInit() {

this.heroes = this.service.getHeroes();

}

selectHero(hero: Hero) { this.selectedHero = hero; }

}

Angular creates, updates, and destroys components as the user moves through the application. Your app can take action at each moment in this lifecycle through optional [lifecycle hooks](https://angular.io/guide/lifecycle-hooks), like ngOnInit().

## **Component metadata**

Metadata

The @[Component](https://angular.io/api/core/Component) decorator identifies the class immediately below it as a component class, and specifies its metadata. In the example code below, you can see that HeroListComponent is just a class, with no special Angular notation or syntax at all. It's not a component until you mark it as one with the @[Component](https://angular.io/api/core/Component)decorator.

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 templatewith the component, either directly with inline code, or by reference. Together, the component and its template describe a view.

In addition to containing or pointing to the template, the @[Component](https://angular.io/api/core/Component) metadata configures, for example, how the component can be referenced in HTML and what services it requires.

Here's an example of basic metadata for HeroListComponent.

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

content\_copy@[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:

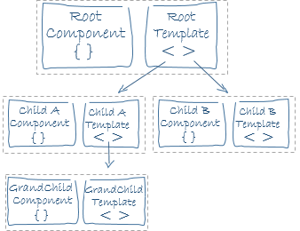
* selector: A CSS selector that tells Angular to create and insert an instance of this component wherever it finds the corresponding tag in template HTML. For example, if an app's HTML contains <app-hero-list></app-hero-list>, then Angular inserts an instance of the HeroListComponent view between those 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](https://angular.io/api/core/Component#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.

## **Templates and views**



You define a component's view with its companion template. A template is a form of HTML that tells Angular how to render the component.

Views are typically arranged hierarchically, allowing you to modify or show and hide entire UI sections or pages as a unit. The template immediately associated with a component defines that component's host view. The component can also define a view hierarchy, which contains embedded views, hosted by other components.



A view hierarchy can include views from components in the same NgModule, but it also can (and often does) include views from components that are defined in different NgModules.

## **Template syntax**

A template looks like regular HTML, except that it also contains Angular [template syntax](https://angular.io/guide/template-syntax), which alters the HTML based on your app's logic and the state of app and DOM data. Your template can use data binding to coordinate the app and DOM data, pipes to transform data before it is displayed, and directives to apply app logic to what gets displayed.

For example, here is a template for the Tutorial's HeroListComponent.

src/app/hero-list.component.html

content\_copy<h2>Hero [List](https://angular.io/api/common/NumberSymbol#List)</h2>

<p><i>Pick [a](https://angular.io/api/router/RouterLinkWithHref) hero from the list</i></p>

<ul>

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

{{hero.name}}

</li>

</ul>

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

This template uses typical HTML elements like <h2> and <p>, and also includes Angular template-syntax elements, \*[ngFor](https://angular.io/api/common/NgForOf), {{hero.name}}, (click), [hero], and <app-hero-detail>. The template-syntax elements tell Angular how to render the HTML to the screen, using program logic and data.

* The \*[ngFor](https://angular.io/api/common/NgForOf) directive tells Angular to iterate over a list.
* {{hero.name}}, (click), and [hero] bind program data to and from the DOM, responding to user input. See more about [data binding](https://angular.io/guide/architecture-components#data-binding) below.
* The <app-hero-detail> tag in the example is an element that represents a new component, HeroDetailComponent.  
  HeroDetailComponent (code not shown) defines the hero-detail child view of HeroListComponent. Notice how custom components like this mix seamlessly with native HTML in the same layouts.

### **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 jQuery 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)

content\_copy<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/template-syntax#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)

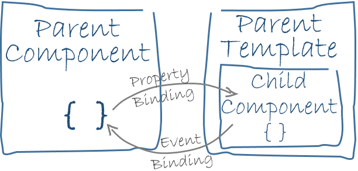
content\_copy<input [([ngModel](https://angular.io/api/forms/NgModel))]="hero.name">

In two-way binding, a data property value flows to the input box from the component as with property binding. The user's changes also flow back to the component, resetting the property to the latest value, as with event binding.

Angular processes all data bindings once for each JavaScript event cycle, from the root of the application component tree through all child components.



Data binding plays an important role in communication between a template and its component, and is also important for communication between parent and child components.



### **Pipes**

Angular pipes let you declare display-value transformations in your template HTML. A class with the @[Pipe](https://angular.io/api/core/Pipe) decorator defines a function that transforms input values to output values for display in a view.

Angular defines various pipes, such as the [date](https://angular.io/api/common/DatePipe) pipe and [currency](https://angular.io/api/common/CurrencyPipe) pipe; for a complete list, see the [Pipes API list](https://angular.io/api?type=pipe). You can also define new pipes.

To specify a value transformation in an HTML template, use the [pipe operator (|)](https://angular.io/guide/template-syntax#pipe).

{{interpolated\_value | pipe\_name}}

You can chain pipes, sending the output of one pipe function to be transformed by another pipe function. A pipe can also take arguments that control how it performs its transformation. For example, you can pass the desired format to the date pipe.

content\_copy<!-- Default format: output 'Jun 15, 2015'-->

<p>Today is {{today | date}}</p>

<!-- fullDate format: output '[Monday](https://angular.io/api/common/WeekDay#Monday), June 15, 2015'-->

<p>The date is {{today | date:'fullDate'}}</p>

<!-- shortTime format: output '9:43 AM'-->

<p>The time is {{today | date:'shortTime'}}</p>

### **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)

content\_copy<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)

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

Angular has more pre-defined directives that either alter the layout structure (for example, [ngSwitch](https://angular.io/guide/template-syntax#ngSwitch)) or modify aspects of DOM elements and components (for example, [ngStyle](https://angular.io/guide/template-syntax#ngStyle) and [ngClass](https://angular.io/guide/template-syntax#ngClass)).

# Introduction to services and dependency injection

Service is a broad category encompassing any value, function, or feature that an app needs. A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well.

Angular distinguishes components from services to increase modularity and reusability. By separating a component's view-related functionality from other kinds of processing, you can make your component classes lean and efficient.

Ideally, a component's job is to enable the user experience and nothing more. A component should present properties and methods for data binding, in order to mediate between the view (rendered by the template) and the application logic (which often includes some notion of a model).

A component can delegate certain tasks to services, such as fetching data from the server, validating user input, or logging directly to the console. By defining such processing tasks in an injectable service class, you make those tasks available to any component. You can also make your app more adaptable by injecting different providers of the same kind of service, as appropriate in different circumstances.

Angular doesn't enforce these principles. Angular does help you follow these principles by making it easy to factor your application logic into services and make those services available to components through dependency injection.

## **Service examples**

Here's an example of a service class that logs to the browser console.

src/app/logger.service.ts (class)

content\_copyexport class Logger {

[log](https://angular.io/api/animations/browser/testing/MockAnimationDriver#log)(msg: any) { console.log(msg); }

error(msg: any) { console.error(msg); }

warn(msg: any) { console.warn(msg); }

}

Services can depend on other services. For example, here's a HeroService that depends on the Logger service, and also uses BackendService to get heroes. That service in turn might depend on the [HttpClient](https://angular.io/api/common/http/HttpClient) service to fetch heroes asynchronously from a server.

src/app/hero.service.ts (class)

content\_copyexport class HeroService {

private heroes: Hero[] = [];

constructor(

private backend: BackendService,

private logger: Logger) { }

getHeroes() {

this.backend.getAll(Hero).then( (heroes: Hero[]) => {

this.logger.log(`Fetched ${heroes.length} heroes.`);

this.heroes.push(...heroes); // fill cache

});

return this.heroes;

}

}

## **Dependency injection (DI)**



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.

* The injector is the main mechanism. Angular creates an application-wide injector for you during the bootstrap process, and additional injectors as needed. You don't have to create injectors.
* An injector creates dependencies, and maintains a container of dependency instances that it reuses if possible.
* A provider is an object that tell an injector how to obtain or create a dependency.

For any dependency that you need in your app, you must register a provider with the app's injector, so that the injector can use the provider to create new instances. For a service, the provider is typically the service class itself.

A dependency doesn't have to be a service—it could be a function, for example, or a value.

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)

content\_copyconstructor(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.

The process of HeroService injection looks something like this.



### **Providing services**

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 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.
* content\_copy@[Injectable](https://angular.io/api/core/Injectable)({
* [providedIn](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 providersproperty of the @[NgModule](https://angular.io/api/core/NgModule)() decorator,
* content\_copy@[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)

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

selector: 'app-hero-list',

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

providers: [ HeroService ]

})

For more detailed information, see the [Dependency Injection](https://angular.io/guide/dependency-injection) section.

# Next steps: tools and techniques

After you understand the basic Angular building blocks, you can begin to learn more about the features and tools that are available to help you develop and deliver Angular applications. Here are some key features.

## **Responsive programming tools**

* [Lifecycle hooks](https://angular.io/guide/lifecycle-hooks): Tap into key moments in the lifetime of a component, from its creation to its destruction, by implementing the lifecycle hook interfaces.
* [Observables and event processing](https://angular.io/guide/observables): How to use observables with components and services to publish and subscribe to messages of any type, such as user-interaction events and asynchronous operation results.

## **Client-server interaction tools**

* [HTTP](https://angular.io/guide/http): Communicate with a server to get data, save data, and invoke server-side actions with an HTTP client.
* [Server-side Rendering](https://angular.io/guide/universal): Angular Universal generates static application pages on the server through server-side rendering (SSR). This allows you to run your Angular app on the server in order to improve performance and show the first page quickly on mobile and low-powered devices, and also facilitate web crawlers.
* [Service Workers](https://angular.io/guide/service-worker-intro): Use a service worker to reduce dependency on the network significantly improving the use experience.

## **Domain-specific libraries**

* [Animations](https://angular.io/guide/animations): Use Angular's animation library to animate component behavior without deep knowledge of animation techniques or CSS.
* [Forms](https://angular.io/guide/forms): Support complex data entry scenarios with HTML-based validation and dirty checking.

## **Support for the development cycle**

* [Testing platform](https://angular.io/guide/testing): Run unit tests on your application parts as they interact with the Angular framework.
* [Internationalization](https://angular.io/guide/i18n): Make your app available in multiple languages with Angular's internationalization (i18n) tools.
* [Compilation](https://angular.io/guide/aot-compiler): Angular provides just-in-time (JIT) compilation for the development environment, and ahead-of-time (AOT) compilation for the production environment.
* [Security guidelines](https://angular.io/guide/security): Learn about Angular's built-in protections against common web-app vulnerabilities and attacks such as cross-site scripting attacks.

## **Setup and deployment tools**

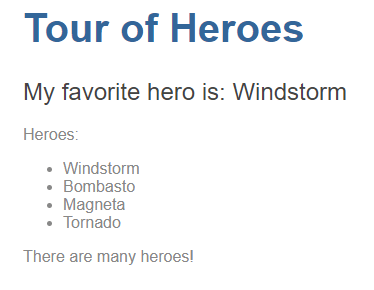
* [Setup for local development](https://angular.io/guide/setup): Set up a new project for development with QuickStart.
* [Installation](https://angular.io/guide/npm-packages): The [Angular CLI](https://cli.angular.io/), Angular applications, and Angular itself depend on features and functionality provided by libraries that are available as [npm](https://docs.npmjs.com/) packages.
* [TypeScript configuration](https://angular.io/guide/typescript-configuration): TypeScript is the primary language for Angular application development.
* [Browser support](https://angular.io/guide/browser-support): Make your apps compatible across a wide range of browsers.
* [Deployment](https://angular.io/guide/deployment): Learn techniques for deploying your Angular application to a remote server.

# Displaying Data

You can display data by binding controls in an HTML template to properties of an Angular component.

In this page, you'll create a component with a list of heroes. You'll display the list of hero names and conditionally show a message below the list.

The final UI looks like this:



## **Showing component properties with interpolation**

The easiest way to display a component property is to bind the property name through interpolation. With interpolation, you put the property name in the view template, enclosed in double curly braces: {{myHero}}.

Follow the [quickstart](https://angular.io/guide/quickstart) instructions for creating a new project named displaying-data.

Delete the app.component.html file. It is not needed for this example.

Then modify the app.component.ts file by changing the template and the body of the component.

When you're done, it should look like this:

src/app/app.component.ts

content\_copy

1. import { [Component](https://angular.io/api/core/Component) } from '@angular/core';
3. @[Component](https://angular.io/api/core/Component)({
4. selector: 'app-root',
5. [template](https://angular.io/api/core/Component#template): `
6. <h1>{{title}}</h1>
7. <h2>My favorite hero is: {{myHero}}</h2>
8. `
9. })
10. export class AppComponent {
11. title = 'Tour of Heroes';
12. myHero = 'Windstorm';
13. }

You added two properties to the formerly empty component: title and myHero.

The template displays the two component properties using double curly brace interpolation:

src/app/app.component.ts (template)

content\_copy[template](https://angular.io/api/core/Component#template): `

<h1>{{title}}</h1>

<h2>My favorite hero is: {{myHero}}</h2>

`

The template is a multi-line string within ECMAScript 2015 backticks (`). The backtick (`)—which is not the same character as a single quote (')—allows you to compose a string over several lines, which makes the HTML more readable.

Angular automatically pulls the value of the title and myHero properties from the component and inserts those values into the browser. Angular updates the display when these properties change.

More precisely, the redisplay occurs after some kind of asynchronous event related to the view, such as a keystroke, a timer completion, or a response to an HTTP request.

Notice that you don't call **new** to create an instance of the AppComponent class. Angular is creating an instance for you. How?

The CSS selector in the @[Component](https://angular.io/api/core/Component) decorator specifies an element named <app-root>. That element is a placeholder in the body of your index.html file:

src/index.html (body)

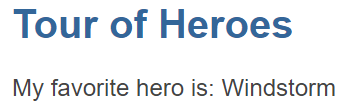
content\_copy<body>

<app-root></app-root>

</body>

When you bootstrap with the AppComponent class (in main.ts), Angular looks for a <app-root> in the index.html, finds it, instantiates an instance of AppComponent, and renders it inside the <app-root> tag.

Now run the app. It should display the title and hero name:



The next few sections review some of the coding choices in the app.

## **Template inline or template file?**

You can store your component's template in one of two places. You can define it inline using the [template](https://angular.io/api/core/Component#template) property, or you can define the template in a separate HTML file and link to it in the component metadata using the @[Component](https://angular.io/api/core/Component) decorator's templateUrl property.

The choice between inline and separate HTML is a matter of taste, circumstances, and organization policy. Here the app uses inline HTML because the template is small and the demo is simpler without the additional HTML file.

In either style, the template data bindings have the same access to the component's properties.

By default, the Angular CLI generates components with a template file. You can override that with:

content\_copyng generate component hero -it

## **Constructor or variable initialization?**

Although this example uses variable assignment to initialize the components, you could instead declare and initialize the properties using a constructor:

content\_copyexport class AppCtorComponent {

title: string;

myHero: string;

constructor() {

this.title = 'Tour of Heroes';

this.myHero = 'Windstorm';

}

}

This app uses more terse "variable assignment" style simply for brevity.

## **Showing an array property with \*ngFor**

To display a list of heroes, begin by adding an array of hero names to the component and redefine myHero to be the first name in the array.

src/app/app.component.ts (class)

content\_copyexport class AppComponent {

title = 'Tour of Heroes';

heroes = ['Windstorm', 'Bombasto', 'Magneta', 'Tornado'];

myHero = this.heroes[0];

}

Now use the Angular [ngFor](https://angular.io/api/common/NgForOf) directive in the template to display each item in the heroes list.

src/app/app.component.ts (template)

content\_copy[template](https://angular.io/api/core/Component#template): `

<h1>{{title}}</h1>

<h2>My favorite hero is: {{myHero}}</h2>

<p>Heroes:</p>

<ul>

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

{{ hero }}

</li>

</ul>

`

This UI uses the HTML unordered list with <ul> and <li> tags. The \*[ngFor](https://angular.io/api/common/NgForOf) in the <li> element is the Angular "repeater" directive. It marks that <li> element (and its children) as the "repeater template":

src/app/app.component.ts (li)

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

{{ hero }}

</li>

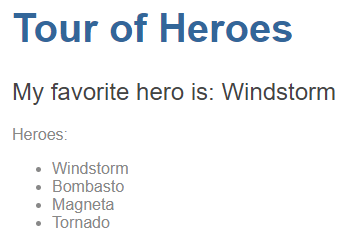
Don't forget the leading asterisk (\*) in \*[ngFor](https://angular.io/api/common/NgForOf). It is an essential part of the syntax. For more information, see the [Template Syntax](https://angular.io/guide/template-syntax#ngFor) page.

Notice the hero in the [ngFor](https://angular.io/api/common/NgForOf) double-quoted instruction; it is an example of a template input variable. Read more about template input variables in the [microsyntax](https://angular.io/guide/template-syntax#microsyntax) section of the [Template Syntax](https://angular.io/guide/template-syntax) page.

Angular duplicates the <li> for each item in the list, setting the hero variable to the item (the hero) in the current iteration. Angular uses that variable as the context for the interpolation in the double curly braces.

In this case, [ngFor](https://angular.io/api/common/NgForOf) is displaying an array, but [ngFor](https://angular.io/api/common/NgForOf) can repeat items for any [iterable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols) object.

Now the heroes appear in an unordered list.



## **Creating a class for the data**

The app's code defines the data directly inside the component, which isn't best practice. In a simple demo, however, it's fine.

At the moment, the binding is to an array of strings. In real applications, most bindings are to more specialized objects.

To convert this binding to use specialized objects, turn the array of hero names into an array of Hero objects. For that you'll need a Hero class:

content\_copyng generate class hero

With the following code:

src/app/hero.ts

content\_copyexport class Hero {

constructor(

public id: number,

public name: string) { }

}

You've defined a class with a constructor and two properties: id and name.

It might not look like the class has properties, but it does. The declaration of the constructor parameters takes advantage of a TypeScript shortcut.

Consider the first parameter:

src/app/hero.ts (id)

content\_copypublic id: number,

That brief syntax does a lot:

* Declares a constructor parameter and its type.
* Declares a public property of the same name.
* Initializes that property with the corresponding argument when creating an instance of the class.

### **Using the Hero class**

After importing the Hero class, the AppComponent.heroes property can return a typed array of Hero objects:

src/app/app.component.ts (heroes)

content\_copyheroes = [

new Hero(1, 'Windstorm'),

new Hero(13, 'Bombasto'),

new Hero(15, 'Magneta'),

new Hero(20, 'Tornado')

];

myHero = this.heroes[0];

Next, update the template. At the moment it displays the hero's id and name. Fix that to display only the hero's name property.

src/app/app.component.ts (template)

content\_copy[template](https://angular.io/api/core/Component#template): `

<h1>{{title}}</h1>

<h2>My favorite hero is: {{myHero.name}}</h2>

<p>Heroes:</p>

<ul>

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

{{ hero.name }}

</li>

</ul>

`

The display looks the same, but the code is clearer.

## **Conditional display with NgIf**

Sometimes an app needs to display a view or a portion of a view only under specific circumstances.

Let's change the example to display a message if there are more than three heroes.

The Angular [ngIf](https://angular.io/api/common/NgIf) directive inserts or removes an element based on a truthy/falsy condition. To see it in action, add the following paragraph at the bottom of the template:

src/app/app.component.ts (message)

content\_copy<p \*[ngIf](https://angular.io/api/common/NgIf)="heroes.length > 3">There are many heroes!</p>

Don't forget the leading asterisk (\*) in \*[ngIf](https://angular.io/api/common/NgIf). It is an essential part of the syntax. Read more about [ngIf](https://angular.io/api/common/NgIf) and \* in the [ngIf section](https://angular.io/guide/template-syntax#ngIf) of the [Template Syntax](https://angular.io/guide/template-syntax) page.

The template expression inside the double quotes, \*[ngIf](https://angular.io/api/common/NgIf)="heroes.length > 3", looks and behaves much like TypeScript. When the component's list of heroes has more than three items, Angular adds the paragraph to the DOM and the message appears. If there are three or fewer items, Angular omits the paragraph, so no message appears. For more information, see the [template expressions](https://angular.io/guide/template-syntax#template-expressions) section of the [Template Syntax](https://angular.io/guide/template-syntax) page.

Angular isn't showing and hiding the message. It is adding and removing the paragraph element from the DOM. That improves performance, especially in larger projects when conditionally including or excluding big chunks of HTML with many data bindings.

Try it out. Because the array has four items, the message should appear. Go back into app.component.ts" and delete or comment out one of the elements from the hero array. The browser should refresh automatically and the message should disappear.

## **Summary**

Now you know how to use:

* **Interpolation** with double curly braces to display a component property.
* **ngFor** to display an array of items.
* A TypeScript class to shape the **model data** for your component and display properties of that model.
* **ngIf** to conditionally display a chunk of HTML based on a boolean expression.