Angular Glossary

Angular has its own vocabulary. Most Angular terms are common English words with a specific meaning within the Angular system.

This glossary lists the most prominent terms and a few less familiar ones that have unusual or unexpected definitions.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

{@a A} {@a aot}

Ahead-of-time (AOT) compilation

You can compile Angular applications at build time. By compiling your application using the compiler-cli, `ngc`, you can bootstrap directly to a module factory, meaning you don't need to include the Angular compiler in your JavaScript bundle. Ahead-of-time compiled applications also benefit from decreased load time and increased performance.

Annotation

In practice, a synonym for Decoration.

{@a attribute-directive}

{@a attribute-directives}

Attribute directives

A category of directive that can listen to and modify the behavior of other HTML elements, attributes, properties, and components. They are usually represented as HTML attributes, hence the name.

For example, you can use the `ngClass` directive to add and remove CSS class names.

Learn about them in the *Attribute Directives* guide.

{@a B}

Barrel

A way to *roll up exports* from several ES2015 modules into a single convenient ES2015 module. The barrel itself is an ES2015 module file that re-exports *selected* exports of other ES2015 modules.

For example, imagine three ES2015 modules in a `heroes` folder:

// heroes/hero.component.ts export class HeroComponent {}

// heroes/hero.model.ts export class Hero {}

// heroes/hero.service.ts export class HeroService {}

Without a barrel, a consumer needs three import statements:

import { HeroComponent } from '../heroes/hero.component.ts'; import { Hero } from '../heroes/hero.model.ts'; import { HeroService } from '../heroes/hero.service.ts';

You can add a barrel to the `heroes` folder (called `index`, by convention) that exports all of these items:

export * from './hero.model.ts'; // re-export all of its exports export * from './hero.service.ts'; // re-export all of its exports export { HeroComponent } from './hero.component.ts'; // re-export the named thing

Now a consumer can import what it needs from the barrel.

import { Hero, HeroService } from '../heroes'; // index is implied

The Angular scoped packages each have a barrel named `index`.

You can often achieve the same result using [NgModules](guide/glossary#ngmodule) instead.

Binding

Usually refers to data binding and the act of binding an HTML object property to a data object property.

Sometimes refers to a dependency-injection binding between a "token"—also referred to as a "key"—and a dependency provider.

Bootstrap

You launch an Angular application by "bootstrapping" it using the application root NgModule (`AppModule`). Bootstrapping identifies an application's top level "root" [component](guide/glossary#component), which is the

first component that is loaded for the application. You can bootstrap multiple apps in the same `index.html`, each app with its own top-level root. {@a C} ## camelCase The practice of writing compound words or phrases such that each word or abbreviation begins with a capital letter _except the first letter, which is lowercase_. Function, property, and method names are typically spelled in camelCase. For example, `square`, `firstName`, and `getHeroes`. Notice that `square` is an example of how you write a single word in camelCase. camelCase is also known as *lower camel case* to distinguish it from *upper camel case*, or [PascalCase](guide/glossary#pascalcase). In Angular documentation, "camelCase" always means *lower camel case*. ## CLI The Angular CLI is a `command line interface` tool that can create a project, add files, and perform a variety of ongoing development tasks such as testing, bundling, and deployment. Learn more in the [Getting Started](guide/quickstart) guide. {@a component} ## Component An Angular class responsible for exposing data to a [view](guide/glossary#view) and handling most of the view's display and user-interaction logic. The *component* is one of the most important building blocks in the Angular system. It is, in fact, an Angular [directive](guide/glossary#directive) with a companion [template](guide/glossary#template). Apply the `@Component` [decorator](guide/glossary#decorator) to the component class, thereby attaching to the class the essential component metadata that Angular needs to create a component instance and render the component with its template as a view. Those familiar with "MVC" and "MVVM" patterns will recognize the component in the role of "controller" or "view model". {@a D} ## dash-case The practice of writing compound words or phrases such that each word is separated by a dash or hyphen (`-`). This form is also known as kebab-case. [Directive](guide/glossary#directive) selectors (like `my-app`) and the root of filenames (such as `hero-list.component.ts`) are often spelled in dash-case. ## Data binding Applications display data values to a user and respond to user actions (such as clicks, touches, and keystrokes). In data binding, you declare the relationship between an HTML widget and data source and let the framework handle the details. Data binding is an alternative to manually pushing application data values into HTML, attaching event listeners, pulling changed values from the screen, and updating application data values. Angular has a rich data-binding framework with a variety of data-binding operations and supporting declaration syntax. Read about the following forms of binding in the [Template Syntax](guide/template-syntax) page: * [Interpolation](guide/template-syntax#interpolation). * [Property binding](guide/template-syntax#property-binding). * [Event binding](guide/template-syntax#event-binding). * [Attribute binding](guide/template-syntax#attribute-binding). * [Class binding](guide/template-syntax#class-binding). * [Style binding](guide/template-syntax#style-binding). * [Two-way data binding with ngModel](guide/template-syntax#ngModel). {@a decorator} {@a decoration} ## Decorator | decoration A *function* that adds metadata to a class, its members (properties, methods) and function arguments. Decorators are an experimental (stage 2), JavaScript language [feature](https://github.com/wycats/javascript-decorators). TypeScript adds support for decorators. To apply a decorator, position it immediately above or to the left of the item it decorates. Angular has its own set of decorators to help it interoperate with your application parts. The following example is a `@Component` decorator that identifies a class as an Angular [component](guide/glossary#component) and an `@Input` decorator applied to the `name` property of that component. The elided object argument to the `@Component` decorator would contain the pertinent component metadata. ``` @Component({...}) export class AppComponent { constructor(@Inject('SpecialFoo') public foo:Foo) {} @Input() name:string; } ``` The scope of a decorator is

limited to the language feature that it decorates. None of the decorations shown here will "leak" to other classes that follow it in the file.
Always include parentheses `()` when applying a decorator.

Dependency injection

A design pattern and mechanism for creating and delivering parts of an application to other parts of an application that request them.

Angular developers prefer to build applications by defining many simple parts that each do one thing well and then wiring them together at runtime.

These parts often rely on other parts. An Angular component part might rely on a service part to get data or perform a calculation. When part "A" relies on another part "B," you say that "A" depends on "B" and that "B" is a dependency of "A."

You can ask a "dependency injection system" to create "A" for us and handle all the dependencies. If "A" needs "B" and "B" needs "C," the system resolves that chain of dependencies and returns a fully prepared instance of "A."

Angular provides and relies upon its own sophisticated dependency-injection system to assemble and run applications by "injecting" application parts into other application parts where and when needed.

At the core, an `injector` returns dependency values on request. The expression `injector.get(token)` returns the value associated with the given token.

A token is an Angular type (`InjectionToken`). You rarely need to work with tokens directly; most methods accept a class name (`Foo`) or a string ("foo") and Angular converts it to a token. When you write `injector.get(Foo)` , the injector returns the value associated with the token for the `Foo` class, typically an instance of `Foo` itself.

During many of its operations, Angular makes similar requests internally, such as when it creates a `component` for display.

The `Injector` maintains an internal map of tokens to dependency values. If the `Injector` can't find a value for a given token, it creates a new value using a `Provider` for that token.

A provider is a recipe for creating new instances of a dependency value associated with a particular token.

An injector can only create a value for a given token if it has a `provider` for that token in its internal provider registry. Registering providers is a critical preparatory step.

Angular registers some of its own providers with every injector. You can register your own providers.

Read more in the Dependency Injection page.

{@a directive}

{@a directives}

Directive

An Angular class responsible for creating, reshaping, and interacting with HTML elements in the browser DOM. The directive is Angular's most fundamental feature.

A directive is usually associated with an HTML element or attribute. This element or attribute is often referred to as the directive itself.

When Angular finds a directive in an HTML template, it creates the matching directive class instance and gives the instance control over that portion of the browser DOM.

You can invent custom HTML markup (for example, `<my-directive>`) to associate with your custom directives. You add this custom markup to HTML templates as if you were writing native HTML. In this way, directives become extensions of HTML itself.

Directives fall into one of the following categories:

- Components combine application logic with an HTML template to render application views. Components are usually represented as HTML elements. They are the building blocks of an Angular application.
- Attribute directives can listen to and modify the behavior of other HTML elements, attributes, properties, and components. They are usually represented as HTML attributes, hence the name.
- Structural directives are responsible for shaping or reshaping HTML layout, typically by adding, removing, or manipulating elements and their children.

{@a E}

ECMAScript

The official JavaScript language specification.

The latest approved version of JavaScript is ECMAScript 2017 (also known as "ES2017" or "ES8"). Many Angular developers write their applications in ES8 or a dialect that strives to be compatible with it, such as

TypeScript.

Most modern browsers only support the much older "ECMAScript 5" (also known as "ES5") standard. Applications written in ES2017, ES2016, ES2015, or one of their dialects must be transpiled to ES5 JavaScript.

Angular developers can write in ES5 directly.

ES2015

Short hand for ECMAScript 2015.

ES5

Short hand for ECMAScript 5, the version of JavaScript run by most modern browsers.

ES6

Short hand for ECMAScript 2015.

{@a F}

{@a G}

{@a H}

{@a I}

Injector

An object in the Angular dependency-injection system that can find a named dependency in its cache or create a dependency with a registered provider.

Input

A directive property that can be the *target* of a property binding (explained in detail in the Template Syntax page). Data values flow *into* this property from the data source identified in the template expression to the right of the equal sign.

See the Input and output properties section of the Template Syntax page.

Interpolation

A form of property data binding in which a template expression between double-curly braces renders as text. That text may be concatenated with neighboring text before it is assigned to an element property or displayed between element tags, as in this example.

My current hero is {{hero.name}}

Read more about interpolation in the Template Syntax page.

{@a J}

{@a jit}

Just-in-time (JIT) compilation

A bootstrapping method of compiling components and modules in the browser and launching the application dynamically. Just-in-time mode is a good choice during development. Consider using the ahead-of-time mode for production apps.

{@a K}

kebab-case

See dash-case.

{@a L}

Lifecycle hooks

Directives and components have a lifecycle managed by Angular as it creates, updates, and destroys them.

You can tap into key moments in that lifecycle by implementing one or more of the lifecycle hook interfaces.

Each interface has a single hook method whose name is the interface name prefixed with `ng`. For example, the `OnInit` interface has a hook method named `ngOnInit`.

Angular calls these hook methods in the following order:

- `ngOnChanges`: when an input/output binding value changes.

- `ngOnInit` : after the first `ngOnChanges` .
- `ngDoCheck` : developer's custom change detection.
- `ngAfterContentInit` : after component content initialized.
- `ngAfterContentChecked` : after every check of component content.
- `ngAfterViewInit` : after a component's views are initialized.
- `ngAfterViewChecked` : after every check of a component's views.
- `ngOnDestroy` : just before the directive is destroyed.

Read more in the Lifecycle Hooks page.

{@a M}

Module

Angular has the following types of modules: * [NgModules](guide/glossary#ngmodule). For details and examples, see the [NgModules](guide/ngmodule) page. * ES2015 modules, as described in this section.

A cohesive block of code dedicated to a single purpose.

Angular apps are modular.

In general, you assemble an application from many modules, both the ones you write and the ones you acquire from others.

A module *exports* something of value in that code, typically one thing such as a class; a module that needs that class *imports* it.

The structure of NgModules and the import/export syntax is based on the ES2015 module standard.

An application that adheres to this standard requires a module loader to load modules on request and resolve inter-module dependencies. Angular doesn't include a module loader and doesn't have a preference for any particular third-party library. You can use any module library that conforms to the standard.

Modules are typically named after the file in which the exported thing is defined. The Angular DatePipe class belongs to a feature module named `date_pipe` in the file `date_pipe.ts` .

You rarely access Angular feature modules directly. You usually import them from an Angular scoped package such as `@angular/core` .

{@a N}

NgModule

Helps you organize an application into cohesive blocks of functionality. An NgModule identifies the components, directives, and pipes that the application uses along with the list of external NgModules that the application needs, such as `FormsModule`. Every Angular application has an application root-module class. By convention, the class is called `AppModule` and resides in a file named `app.module.ts`. For details and examples, see [NgModules](guide/ngmodule).

{@a O}

Observable

An array whose items arrive asynchronously over time. Observables help you manage asynchronous data, such as data coming from a backend service. Observables are used within Angular itself, including Angular's event system and its HTTP client service.

To use observables, Angular uses a third-party library called Reactive Extensions (RxJS). Observables are a proposed feature for ES2016, the next version of JavaScript.

Output

A directive property that can be the *target* of event binding (read more in the event binding section of the Template Syntax page). Events stream *out* of this property to the receiver identified in the template expression to the right of the equal sign.

See the Input and output properties section of the Template Syntax page.

{@a P}

PascalCase

The practice of writing individual words, compound words, or phrases such that each word or abbreviation begins with a capital letter. Class names are typically spelled in PascalCase. For example, `Person` and `HeroDetailComponent`.

This form is also known as *upper camel case* to distinguish it from *lower camel case* or simply camelCase. In this documentation, "PascalCase" means *upper camel case* and "camelCase" means *lower camel case*.

Pipe

An Angular pipe is a function that transforms input values to output values for display in a view. Here's an

example that uses the built-in `currency` pipe to display a numeric value in the local currency.

Price: {{product.price | currency}}

You can also write your own custom pipes. Read more in the page on pipes.

Provider

A *provider* creates a new instance of a dependency for the dependency injection system. It relates a lookup token to code—sometimes called a "recipe"—that can create a dependency value.

{@a Q}

{@a R}

Reactive forms

A technique for building Angular forms through code in a component. The alternative technique is template-driven forms.

When building reactive forms:

- The "source of truth" is the component. The validation is defined using code in the component.
- Each control is explicitly created in the component class with `new FormControl()` or with `FormBuilder`.
- The template input elements do *not* use `ngModel`.
- The associated Angular directives are all prefixed with `Form`, such as `FormGroup`, `FormControl`, and `FormControlName`.

Reactive forms are powerful, flexible, and a good choice for more complex data-entry form scenarios, such as dynamic generation of form controls.

Router

Most applications consist of many screens or views. The user navigates among them by clicking links and buttons, and performing other similar actions that cause the application to replace one view with another.

The Angular component router is a richly featured mechanism for configuring and managing the entire view navigation process, including the creation and destruction of views.

In most cases, components become attached to a router by means of a `RouterConfig` that defines routes

to views.

A routing component's template has a `RouterOutlet` element where it can display views produced by the router.

Other views in the application likely have anchor tags or buttons with `RouterLink` directives that users can click to navigate.

For more information, see the Routing & Navigation page.

Router module

A separate NgModule that provides the necessary service providers and directives for navigating through application views.

For more information, see the Routing & Navigation page.

Routing component

An Angular component with a `RouterOutlet` that displays views based on router navigations.

For more information, see the Routing & Navigation page.

{@a S}

Scoped package

A way to group related *npm* packages. Read more at the npm-scope page.

NgModules are delivered within *scoped packages* such as `@angular/core`, `@angular/common`, `@angular/platform-browser-dynamic`, `@angular/http`, and `@angular/router`.

Import a scoped package the same way that you import a normal package. The only difference, from a consumer perspective, is that the scoped package name begins with the Angular *scope name*, `@angular`.

Service

For data or logic that is not associated with a specific view or that you want to share across components, build services.

Applications often require services such as a hero data service or a logging service.

A service is a class with a focused purpose. You often create a service to implement features that are independent from any specific view, provide shared data or logic across components, or encapsulate external interactions.

Applications often require services such as a data service or a logging service.

For more information, see the Services page of the Tour of Heroes tutorial.

{@a snake-case}

snake_case

The practice of writing compound words or phrases such that an underscore (`_`) separates one word from the next. This form is also known as *underscore case*.

{@a structural-directive}

{@a structural-directives}

Structural directives

A category of directive that can shape or reshape HTML layout, typically by adding and removing elements in the DOM. The `ngIf` "conditional element" directive and the `ngFor` "repeater" directive are well-known examples.

Read more in the Structural Directives page.

{@a T}

Template

A chunk of HTML that Angular uses to render a view with the support and guidance of an Angular directive, most notably a component.

Template-driven forms

A technique for building Angular forms using HTML forms and input elements in the view. The alternate technique is Reactive Forms.

When building template-driven forms:

- The "source of truth" is the template. The validation is defined using attributes on the individual input elements.
- Two-way binding with `ngModel` keeps the component model synchronized with the user's entry into the input elements.
- Behind the scenes, Angular creates a new control for each input element, provided you have set up a `name` attribute and two-way binding for each input.
- The associated Angular directives are all prefixed with `ng` such as `ngForm`, `ngModel`, and `ngModelGroup`.

Template-driven forms are convenient, quick, and simple. They are a good choice for many basic data-entry form scenarios.

Read about how to build template-driven forms in the Forms page.

Template expression

A TypeScript-like syntax that Angular evaluates within a data binding.

Read about how to write template expressions in the Template expressions section of the Template Syntax page.

Transpile

The process of transforming code written in one form of JavaScript (such as TypeScript) into another form of JavaScript (such as ES5).

TypeScript

A version of JavaScript that supports most ECMAScript 2015 language features such as decorators.

TypeScript is also notable for its optional typing system, which provides compile-time type checking and strong tooling support (such as "intellisense," code completion, refactoring, and intelligent search). Many code editors and IDEs support TypeScript either natively or with plugins.

TypeScript is the preferred language for Angular development, although you can use other JavaScript dialects such as ES5.

Read more about TypeScript at typescriptlang.org.

{@a U}

{@a V}

View

A portion of the screen that displays information and responds to user actions such as clicks, mouse moves, and keystrokes.

Angular renders a view under the control of one or more directives, especially component directives and their companion templates. The component plays such a prominent role that it's often convenient to refer to a component as a view.

Views often contain other views. Any view might be loaded and unloaded dynamically as the user navigates through the application, typically under the control of a router.

{@a W}

{@a X}

{@a Y}

{@a Z}

Zone

A mechanism for encapsulating and intercepting a JavaScript application's asynchronous activity.

The browser DOM and JavaScript have a limited number of asynchronous activities, such as DOM events (for example, clicks), promises, and XHR calls to remote servers.

Zones intercept all of these activities and give a "zone client" the opportunity to take action before and after the async activity finishes.

Angular runs your application in a zone where it can respond to asynchronous events by checking for data changes and updating the information it displays via data bindings.

Learn more about zones in this Brian Ford video.