Glossary

Angular has its own vocabulary. Most Angular terms are common English words or computing terms that have a specific meaning within the Angular system.

This glossary lists the most prominent terms and a few less familiar ones with 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

The Angular ahead-of-time (AOT) compiler converts Angular HTML and TypeScript code into efficient JavaScript code during the build phase, before the browser downloads and runs that code. This is the best compilation mode for production environments, with decreased load time and increased performance compared to just-in-time (JIT) compilation.

By compiling your application using the `ngc` command-line tool, you can bootstrap directly to a module factory, so you don't need to include the Angular compiler in your JavaScript bundle.

{@a angular-element}

Angular element

An Angular component packaged as a custom element.

Learn more in Angular Elements Overview.

{@a apf}

Angular package format (APF)

An Angular specific specification for layout of npm packages that is used by all first-party Angular packages, and most third-party Angular libraries.

Learn more in the Angular Package Format specification.

{@a annotation}

annotation

A structure that provides metadata for a class. See decorator.

{@a app-shell}

app-shell

App shell is a way to render a portion of your application using a route at build time. This gives users a meaningful first paint of your application that appears quickly because the browser can render static HTML and CSS without the need to initialize JavaScript.

Learn more in The App Shell Model.

You can use the Angular CLI to generate an app shell. This can improve the user experience by quickly launching a static rendered page (a skeleton common to all pages) while the browser downloads the full client version and switches to it automatically after the code loads.

See also Service Worker and PWA. {@a architect}

Architect

The tool that the CLI uses to perform complex tasks such as compilation and test running, according to a provided configuration. Architect is a shell that runs a builder (defined in an npm package) with a given target configuration.

In the workspace configuration file, an "architect" section provides configuration options for Architect builders.

For example, a built-in builder for linting is defined in the package `@angular-devkit/build_angular:tslint`, which uses the TSLint tool to perform linting, with a configuration specified in a `tslint.json` file.

Use the CLI command `ng run` to invoke a builder by specifying a target configuration associated with that builder. Integrators can add builders to enable tools and workflows to run through the Angular CLI. For example, a custom builder can replace the third-party tools used by the built-in implementations for CLI commands such as `ng build` or `ng test`.

{@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.

Learn more in Attribute Directives.

{@a B}

{@a binding}

binding

Generally, the practice of setting a variable or property to a data value. Within Angular, typically refers to data binding, which coordinates DOM object properties with data object properties.

Sometimes refers to a dependency-injection binding between a token and a dependency provider.

{@a bootstrap}

bootstrap

A way to initialize and launch an application or system.

In Angular, an application's root NgModule (`AppModule`) has a `bootstrap` property that identifies the application's top-level components. During the bootstrap process, Angular creates and inserts these components into the `index.html` host web page. You can bootstrap multiple applications in the same `index.html`. Each application contains its own components.

Learn more in Bootstrapping.

{@a builder}

builder

A function that uses the Architect API to perform a complex process such as "build" or "test". The builder code is defined in an npm package.

For example, BrowserBuilder runs a webpack build for a browser target and KarmaBuilder starts the Karma server and runs a webpack build for unit tests.

The CLI command `ng run` invokes a builder with a specific target configuration. The workspace configuration file, `angular.json`, contains default configurations for built-in builders.

{@a C}

{@a case-conventions} {@a dash-case} {@a camelcase} {@a kebab-case}

case types

Angular uses capitalization conventions to distinguish the names of various types, as described in the naming guidelines section of the Style Guide. Here's a summary of the case types:

- camelCase : Symbols, properties, methods, pipe names, non-component directive selectors, constants. Standard or lower camel case uses lowercase on the first letter of the item. For example, "selectedHero".
- UpperCamelCase (or PascalCase): Class names, including classes that define components, interfaces, NgModules, directives, and pipes, Upper camel case uses uppercase on the first letter of the item. For example, "HeroListComponent".
- dash-case (or "kebab-case"): Descriptive part of file names, component selectors. For example, "app-hero-list".
- underscore_case (or "snake_case"): Not typically used in Angular. Snake case uses words connected with underscores. For example, "convert_link_mode".
- UPPER_UNDERSCORE_CASE (or UPPER_SNAKE_CASE, or SCREAMING_SNAKE_CASE): Traditional for constants (acceptable, but prefer camelCase). Upper snake case uses words in all capital letters connected with underscores. For example, "FIX_ME".

{@a change-detection}

change detection

The mechanism by which the Angular framework synchronizes the state of an application's UI with the state of the data. The change detector checks the current state of the data model whenever it runs, and maintains it as the previous state to compare on the next iteration.

As the application logic updates component data, values that are bound to DOM properties in the view can change. The change detector is responsible for updating the view to reflect the current data model. Similarly, the user can interact with the UI, causing events that change the state of the data model. These events can trigger change detection.

Using the default change-detection strategy, the change detector goes through the view hierarchy on each VM turn to check every data-bound property in the template. In the first phase, it compares the current state of the dependent data with the previous state, and collects changes. In the second phase, it updates the page DOM to reflect any new data values.

If you set the `OnPush` change-detection strategy, the change detector runs only when explicitly invoked, or when it is triggered by an `Input` reference change or event handler. This typically improves performance. For more information, see Optimize Angular's change detection.

{@a class-decorator}

class decorator

A decorator that appears immediately before a class definition, which declares the class to be of the given type, and provides metadata suitable to the type.

The following decorators can declare Angular class types:

- `@Component()`
- `@Directive()`
- `@Pipe()`
- `@Injectable()`
- `@NgModule()`

{@a class-field-decorator}

class field decorator

A decorator statement immediately before a field in a class definition that declares the type of that field. Some examples are `@Input` and `@Output`.

{@a collection}

collection

In Angular, a set of related schematics collected in an npm package.

{@a cli}

command-line interface (CLI)

The Angular CLI is a command-line tool for managing the Angular development cycle. Use it to create the initial filesystem scaffolding for a workspace or project, and to run schematics that add and modify code for initial generic versions of various elements. The CLI supports all stages of the development cycle, including building, testing, bundling, and deployment.

- To begin using the CLI for a new project, see Local Environment Setup.
- To learn more about the full capabilities of the CLI, see the CLI command reference.

See also Schematics CLI.

{@a component}

component

A class with the `@Component()` decorator that associates it with a companion template. Together, the component class and template define a view. A component is a special type of directive. The `@Component()` decorator extends the `@Directive()` decorator with template-oriented features.

An Angular component class is responsible for exposing data and handling most of the view's display and user-interaction logic through data binding.

Read more about component classes, templates, and views in Introduction to Angular concepts.

configuration

See workspace configuration

{@a content-projection}

content projection

A way to insert DOM content from outside a component into the component's view in a designated spot.

For more information, see Responding to changes in content.

{@a custom-element}

custom element

A web platform feature, currently supported by most browsers and available in other browsers through polyfills (see Browser support).

The custom element feature extends HTML by allowing you to define a tag whose content is created and controlled by JavaScript code. A custom element (also called a *web component*) is recognized by a browser when it's added to the CustomElementRegistry.

You can use the API to transform an Angular component so that it can be registered with the browser and used in any HTML that you add directly to the DOM within an Angular application. The custom element tag inserts the component's view, with change-detection and data-binding functionality, into content that would otherwise be displayed without Angular processing.

See Angular element.

See also dynamic component loading.

{@a D}

{@a data-binding}

data binding

A process that allows applications to 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 a 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.

Read about the following forms of binding in Angular's Template Syntax:

- Interpolation
- Property binding
- Event binding
- Attribute binding
- Class binding
- Style binding
- Two-way data binding with ngModel

{@a declarable}

declarable

A class type that you can add to the `declarations` list of an NgModule. You can declare components, directives, and pipes.

Don't declare the following:

- A class that's already declared in another NgModule
- An array of directives imported from another package. For example, don't declare `FORMS_DIRECTIVES` from `@angular/forms`
- NgModule classes
- Service classes
- Non-Angular classes and objects, such as strings, numbers, functions, entity models, configurations, business logic, and helper classes

{@a decorator}

{@a decoration}

decorator | decoration

A function that modifies a class or property definition. Decorators (also called *annotations*) are an experimental (stage 2) JavaScript language feature. TypeScript adds support for decorators.

Angular defines decorators that attach metadata to classes or properties so that it knows what those classes or properties mean and how they should work.

See class decorator, class field decorator.

{@a di}

{@a dependency-injection}

dependency injection (DI)

A design pattern and mechanism for creating and delivering some parts of an application (dependencies) to other parts of an application that require them.

In Angular, dependencies are typically services, but they also can be values, such as strings or functions. An injector for an application (created automatically during bootstrap) instantiates dependencies when needed, using a configured provider of the service or value.

Learn more in Dependency Injection in Angular.

{@a di-token}

DI token

A lookup token associated with a dependency provider, for use with the dependency injection system.

{@a directive} {@a directives}

directive

A class that can modify the structure of the DOM or modify attributes in the DOM and component data model. A directive class definition is immediately preceded by a `@Directive()` decorator that supplies metadata.

A directive class is usually associated with an HTML element or attribute, and that 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.

There are three categories of directive:

- Components use `@Component()` (an extension of `@Directive()`) to associate a template with a class.
- Attribute directives modify behavior and appearance of page elements.
- Structural directives modify the structure of the DOM.

Angular supplies a number of built-in directives that begin with the `ng` prefix. You can also create new directives to implement your own functionality. You associate a *selector* (an HTML tag such as `<my-directive>`) with a custom directive; this extends the template syntax that you can use in your applications.

UpperCamelCase, such as `NgIf`, refers to a directive class. You can use **UpperCamelCase** when describing properties and directive behavior.

lowerCamelCase, such as `ngIf` refers to a directive's attribute name. You can use **lowerCamelCase** when describing how to apply the directive to an element in the HTML template.

{@a dom}

domain-specific language (DSL)

A special-purpose library or API; see Domain-specific language. Angular extends TypeScript with domain-specific languages for a number of domains relevant to Angular applications, defined in NgModules such as animations, forms, and routing and navigation.

{@a dynamic-components}

dynamic component loading

A technique for adding a component to the DOM at run time. Requires that you exclude the component from compilation and then connect it to Angular's change-detection and event-handling framework when you add it to the DOM.

See also custom element, which provides an easier path with the same result.

{@a E}

{@a eager-loading}

eager loading

NgModules or components that are loaded on launch are called eager-loaded, to distinguish them from those that are loaded at run time (lazy-loaded). See lazy loading.

{@a ecma}

ECMAScript

The official JavaScript language specification.

Not all browsers support the latest ECMAScript standard, but you can use a transpiler (like TypeScript) to write code using the latest features, which will then be transpiled to code that runs on versions that are supported by browsers.

To learn more, see Browser Support.

{@a element}

element

Angular defines an `ElementRef` class to wrap render-specific native UI elements. In most cases, this allows you to use Angular templates and data binding to access DOM elements without reference to the native element.

The documentation generally refers to *elements* (`ElementRef` instances), as distinct from *DOM elements* (which can be accessed directly if necessary).

Compare to custom element.

{@a entry-point}

entry point

A JavaScript module that is intended to be imported by a user of an npm package. An entry-point module typically re-exports symbols from other internal modules. A package can contain multiple entry points. For example, the `@angular/core` package has two entry-point modules, which can be imported using the module names `@angular/core` and `@angular/core/testing` .

{@a F}

{@a form-control}

form control

A instance of `FormControl` , which is a fundamental building block for Angular forms. Together with `FormGroup` and `FormArray` , tracks the value, validation, and status of a form input element.

Read more forms in the Introduction to forms in Angular.

{@a form-model}

form model

The "source of truth" for the value and validation status of a form input element at a given point in time. When using reactive forms, the form model is created explicitly in the component class. When using template-driven forms, the form model is implicitly created by directives.

Learn more about reactive and template-driven forms in the Introduction to forms in Angular.

{@a form-validation}

form validation

A check that runs when form values change and reports whether the given values are correct and complete, according to the defined constraints. Reactive forms apply validator functions. Template-driven forms use validator directives.

To learn more, see Form Validation.

{@a G}

{@a H}

{@a I}

{@a immutability}

immutability

The inability to alter the state of a value after its creation. Reactive forms perform immutable changes in that each change to the data model produces a new data model rather than modifying the existing one. Template-driven forms perform mutable changes with `NgModel` and two-way data binding to modify the existing data model in place.

{@a injectable}

injectable

An Angular class or other definition that provides a dependency using the dependency injection mechanism. An injectable service class must be marked by the `@Injectable()` decorator. Other items, such as constant values, can also be injectable.

{@a injector}

injector

An object in the Angular dependency-injection system that can find a named dependency in its cache or create a dependency using a configured provider. Injectors are created for NgModules automatically as part of the bootstrap process and are inherited through the component hierarchy.

- An injector provides a singleton instance of a dependency, and can inject this same instance in multiple components.
- A hierarchy of injectors at the NgModule and component level can provide different instances of a dependency to their own components and child components.
- You can configure injectors with different providers that can provide different implementations of the same dependency.

Learn more about the injector hierarchy in Hierarchical Dependency Injectors.

{@a input}

input

When defining a directive, the `@Input()` decorator on a directive property makes that property available as a *target* of a property binding. Data values flow into an input property from the data source identified in the template expression to the right of the equal sign.

To learn more, see input and output properties.

{@a interpolation}

interpolation

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

```
<label>My current hero is {{hero.name}}</label>
```

Read more in the Interpolation guide.

{@a ivy}

Ivy

Ivy is the historical code name for Angular's current compilation and rendering pipeline. It is now the only supported engine, so everything uses Ivy.

{@a J}

{@a javascript}

JavaScript

See ECMAScript, TypeScript.

{@a jit}

just-in-time (JIT) compilation

The Angular just-in-time (JIT) compiler converts your Angular HTML and TypeScript code into efficient JavaScript code at run time, as part of bootstrapping.

JIT compilation is the default (as opposed to AOT compilation) when you run Angular's `ng build` and `ng serve` CLI commands, and is a good choice during development. JIT mode is strongly discouraged for production use because it results in large application payloads that hinder the bootstrap performance.

Compare to ahead-of-time (AOT) compilation.

{@a K}

{@a L}

{@a lazy-load}

lazy loading

A process that speeds up application load time by splitting the application into multiple bundles and loading them on demand. For example, dependencies can be lazy loaded as needed—as opposed to eager-loaded modules that are required by the root module and are thus loaded on launch.

The router makes use of lazy loading to load child views only when the parent view is activated. Similarly, you can build custom elements that can be loaded into an Angular application when needed.

{@a library}

library

In Angular, a project that provides functionality that can be included in other Angular applications. A library isn't a complete Angular application and can't run independently. (To add re-usable Angular functionality to non-Angular web applications, you can use Angular custom elements.)

- Library developers can use the Angular CLI to `generate` scaffolding for a new library in an existing workspace, and can publish a library as an `npm` package.
- Application developers can use the Angular CLI to `add` a published library for use with an application in the same workspace.

See also schematic.

{@a lifecycle-hook}

lifecycle hook

An interface that allows you to tap into the lifecycle of directives and components as they are created, updated, and destroyed.

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 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.

To learn more, see Lifecycle Hooks.

{@a M}

{@a module}

module

In general, a module collects a block of code dedicated to a single purpose. Angular uses standard JavaScript modules and also defines an Angular module, `NgModule`.

In JavaScript (ECMAScript), each file is a module and all objects defined in the file belong to that module. Objects can be exported, making them public, and public objects can be imported for use by other modules.

Angular ships as a collection of JavaScript modules (also called libraries). Each Angular library name begins with the `@angular` prefix. Install Angular libraries with the npm package manager and import parts of them with JavaScript `import` declarations.

Compare to NgModule.

{@a N}

{@a ngcc}

ngcc

Angular compatibility compiler. If you build your application using Ivy, but it depends on libraries that have not been compiled with Ivy, the CLI uses `ngcc` to automatically update the dependent libraries to use Ivy.

{@a ngmodule}

NgModule

A class definition preceded by the `@NgModule()` decorator, which declares and serves as a manifest for a block of code dedicated to an application domain, a workflow, or a closely related set of capabilities.

Like a JavaScript module, an NgModule can export functionality for use by other NgModules and import public functionality from other NgModules. The metadata for an NgModule class collects components, directives, and pipes that the application uses along with the list of imports and exports. See also declarable.

NgModules are typically named after the file in which the exported thing is defined. For example, the Angular DatePipe class belongs to a feature module named `date_pipe` in the file `date_pipe.ts`. You import them from an Angular scoped package such as `@angular/core`.

Every Angular application has a root module. By convention, the class is called `AppModule` and resides in a file named `app.module.ts`.

To learn more, see NgModules.

{@a npm-package}

npm package

The npm package manager is used to distribute and load Angular modules and libraries.

Learn more about how Angular uses Npm Packages.

{@a ngc}

ngc

`ngc` is a Typescript-to-Javascript transpiler that processes Angular decorators, metadata, and templates, and emits JavaScript code. The most recent implementation is internally referred to as `ngtsc` because it's a minimalistic wrapper around the TypeScript compiler `tsc` that adds a transform for processing Angular code.

{@a O}

{@a observable}

observable

A producer of multiple values, which it pushes to subscribers. Used for asynchronous event handling throughout Angular. You execute an observable by subscribing to it with its `subscribe()` method, passing callbacks for notifications of new values, errors, or completion.

Observables can deliver single or multiple values of any type to subscribers, either synchronously (as a function delivers a value to its caller) or on a schedule. A subscriber receives notification of new values as they are produced and notification of either normal completion or error completion.

Angular uses a third-party library called Reactive Extensions (RxJS).

To learn more, see Observables.

{@a observer}

observer

An object passed to the `subscribe()` method for an observable. The object defines the callbacks for the subscriber.

{@a output}

output

When defining a directive, the `@Output{}` decorator on a directive property makes that property available as a *target* of event binding. Events stream *out* of this property to the receiver identified in the template expression to the right of the equal sign.

To learn more, see Input and Output Properties.

{@a P}

{@a pipe}

pipe

A class which is preceded by the `@Pipe{}` decorator and which defines a function that transforms input values to output values for display in a view. Angular defines various pipes, and you can define new pipes.

To learn more, see Pipes.

{@a platform}

platform

In Angular terminology, a platform is the context in which an Angular application runs. The most common platform for Angular applications is a web browser, but it can also be an operating system for a mobile device, or a web server.

Support for the various Angular run-time platforms is provided by the `@angular/platform-*` packages. These packages allow applications that make use of `@angular/core` and `@angular/common` to execute in different environments by providing implementation for gathering user input and rendering UIs for the given platform. Isolating platform-specific functionality allows the developer to make platform-independent use of the rest of the framework.

- When running in a web browser, `BrowserModule` is imported from the `platform-browser` package, and supports services that simplify security and event processing, and allows applications to access browser-specific features, such as interpreting keyboard input and controlling the title of the document being displayed. All applications running in the browser use the same platform service.
- When server-side rendering (SSR) is used, the `platform-server` package provides web server implementations of the `DOM`, `XMLHttpRequest`, and other low-level features that don't rely on a browser.

{@a polyfill}

polyfill

An npm package that plugs gaps in a browser's JavaScript implementation. See Browser Support for polyfills that support particular functionality for particular platforms.

{@a project}

project

In the Angular CLI, a standalone application or library that can be created or modified by a CLI command.

A project, as generated by the `ng new`, contains the set of source files, resources, and configuration files that you need to develop and test the application using the CLI. Projects can also be created with the `ng generate application` and `ng generate library` commands.

For more information, see Project File Structure.

The `angular.json` file configures all projects in a workspace.

{@a provider}

provider

An object that implements one of the `Provider` interfaces. A provider object defines how to obtain an injectable dependency associated with a DI token. An injector uses the provider to create a new instance of a dependency for a class that requires it.

Angular registers its own providers with every injector, for services that Angular defines. You can register your own providers for services that your application needs.

See also service, dependency injection.

Learn more in Dependency Injection.

{@a Q}

{@a R}

{@a reactive-forms}

reactive forms

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

When using reactive forms:

- The "source of truth", the form model, is defined in the component class.
- Validation is set up through validation functions rather than validation directives.
- Each control is explicitly created in the component class by creating a `FormControl` instance manually or with `FormBuilder`.
- The template input elements do *not* use `ngModel`.
- The associated Angular directives are prefixed with `form`, such as `formControl`, `formGroup`, and `formControlName`.

The alternative is a template-driven form. For an introduction and comparison of both forms approaches, see Introduction to Angular Forms.

{@a resolver}

resolver

A class that implements the Resolve interface (or a function with the same signature as the resolve() method) that you use to produce or retrieve data that is needed before navigation to a requested route can be completed.

Resolvers run after all route guards for a route tree have been executed and have succeeded.

See an example of using a resolve guard to retrieve dynamic data.

{@a route-guard}

route guard

A method that controls navigation to a requested route in a routing application. Guards determine whether a route can be activated or deactivated, and whether a lazy-loaded module can be loaded.

Learn more in the Routing and Navigation guide.

{@a router} {@a router-module}

router

A tool that configures and implements navigation among states and views within an Angular application.

The `Router` module is an NgModule that provides the necessary service providers and directives for navigating through application views. A routing component is one that imports the `Router` module and whose template contains a `RouterOutlet` element where it can display views produced by the router.

The router defines navigation among views on a single page, as opposed to navigation among pages. It interprets URL-like links to determine which views to create or destroy, and which components to load or unload. It allows you to take advantage of lazy loading in your Angular applications.

To learn more, see Routing and Navigation.

{@a router-outlet}

router outlet

A directive that acts as a placeholder in a routing component's template. Angular dynamically renders the template based on the current router state.

{@a router-component}

routing component

An Angular component with a `RouterOutlet` directive in its template that displays views based on router navigations.

For more information, see Routing and Navigation.

{@a rule}

rule

In schematics, a function that operates on a file tree to create, delete, or modify files in a specific manner.

{@a S}

{@a schematic}

schematic

A scaffolding library that defines how to generate or transform a programming project by creating, modifying, refactoring, or moving files and code. A schematic defines rules that operate on a virtual file system called a tree.

The Angular CLI uses schematics to generate and modify Angular projects and parts of projects.

- Angular provides a set of schematics for use with the CLI. See the Angular CLI command reference. The `ng add` command runs schematics as part of adding a library to your project. The `ng generate` command runs schematics to create applications, libraries, and Angular code constructs.
- Library developers can create schematics that enable the Angular CLI to add and update their published libraries, and to generate artifacts the library defines. Add these schematics to the npm package that you use to publish and share your library.

For more information, see Schematics and Integrating Libraries with the CLI.

{@a schematics-cli}

Schematics CLI

Schematics come with their own command-line tool. Using Node 6.9 or above, install the Schematics CLI globally:

npm install -g @angular-devkit/schematics-cli

This installs the `schematics` executable, which you can use to create a new schematics collection with an initial named schematic. The collection folder is a workspace for schematics. You can also use the `schematics` command to add a new schematic to an existing collection, or extend an existing schematic.

{@a scoped-package}

scoped package

A way to group related npm packages. NgModules are delivered within scoped packages whose names begin with the Angular *scope name* `@angular`. For example, `@angular/core`, `@angular/common`, `@angular/forms`, and `@angular/router`.

Import a scoped package in the same way that you import a normal package.

{@a server-side-rendering}

server-side rendering

A technique that generates static application pages on the server, and can generate and serve those pages in response to requests from browsers. It can also pre-generate pages as HTML files that you serve later.

This technique can improve performance on mobile and low-powered devices and improve the user experience by showing a static first page quickly while the client-side application is loading. The static version can also make your application more visible to web crawlers.

You can easily prepare an application for server-side rendering by using the CLI to run the Angular Universal tool, using the `@nguniversal/express-engine` schematic.

{@a service}

service

In Angular, a class with the @Injectable() decorator that encapsulates non-UI logic and code that can be reused across an application. Angular distinguishes components from services to increase modularity and reusability.

The `@Injectable()` metadata allows the service class to be used with the dependency injection mechanism. The injectable class is instantiated by a provider. Injectors maintain lists of providers and use them to provide service instances when they are required by components or other services.

To learn more, see Introduction to Services and Dependency Injection.

{@a structural-directive} {@a structural-directives}

structural directives

A category of directive that is responsible for shaping HTML layout by modifying the DOM—that is, adding, removing, or manipulating elements and their children.

To learn more, see Structural Directives.

{@a subscriber}

subscriber

A function that defines how to obtain or generate values or messages to be published. This function is executed when a consumer calls the `subscribe()` method of an observable.

The act of subscribing to an observable triggers its execution, associates callbacks with it, and creates a `Subscription` object that lets you unsubscribe.

The `subscribe()` method takes a JavaScript object (called an observer) with up to three callbacks, one for each type of notification that an observable can deliver:

- The `next` notification sends a value such as a number, a string, or an object.
- The `error` notification sends a JavaScript Error or exception.
- The `complete` notification doesn't send a value, but the handler is called when the call completes. Scheduled values can continue to be returned after the call completes.

{@a T}

{@a target}

target

A buildable or runnable subset of a project, configured as an object in the workspace configuration file, and executed by an Architect builder.

In the `angular.json` file, each project has an "architect" section that contains targets which configure builders. Some of these targets correspond to CLI commands, such as `build`, `serve`, `test`, and `lint`.

For example, the Architect builder invoked by the `ng build` command to compile a project uses a particular build tool, and has a default configuration with values that you can override on the command line. The `build` target also defines an alternate configuration for a "development" build, which you can invoke with the `--configuration development` flag on the `build` command.

The Architect tool provides a set of builders. The `ng new` command provides a set of targets for the initial application project. The `ng generate application` and `ng generate library` commands provide a set of targets for each new project. These targets, their options and configurations, can be customized to meet the needs of your project. For example, you may want to add a "staging" or "testing" configuration to a project's "build" target.

You can also define a custom builder, and add a target to the project configuration that uses your custom builder. You can then run the target using the `ng run` CLI command.

{@a template}

template

Code that defines how to render a component's view.

A template combines straight HTML with Angular data-binding syntax, directives, and template expressions (logical constructs). The Angular elements insert or calculate values that modify the HTML elements before the page is displayed. Learn more about Angular template language in the Template Syntax guide.

A template is associated with a component class through the `@Component()` decorator. The template code can be provided inline, as the value of the `template` property, or in a separate HTML file linked through the `templateUrl` property.

Additional templates, represented by `TemplateRef` objects, can define alternative or *embedded* views, which can be referenced from multiple components.

{@a template-driven-forms}

template-driven forms

A format for building Angular forms using HTML forms and input elements in the view. The alternative format uses the reactive forms framework.

When using 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 prefixed with `ng` such as `ngForm`, `ngModel`, and `ngModelGroup`.

The alternative is a reactive form. For an introduction and comparison of both forms approaches, see Introduction to Angular Forms.

{@a template-expression}

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 Interpolation guide.

{@a template-reference-variable}

template reference variable

A variable defined in a template that references an instance associated with an element, such as a directive instance, component instance, template as in `TemplateRef`, or DOM element. After declaring a template reference variable on an element in a template, you can access values from that variable elsewhere within the same template. The following example defines a template reference variable named `#phone`.

For more information, see the Template reference variable guide.

{@a template-input-variable}

template input variable

A template input variable is a variable you can reference within a single instance of the template. You declare a template input variable using the `let` keyword as in `let customer`.

```
<tr *ngFor="let customer of customers;">
    <td>{{customer.customerNo}}</td>
    <td>{{customer.name}}</td>
```

```
      <td>{{customer.address}}</td>
      <td>{{customer.city}}</td>
      <td>{{customer.state}}</td>
      <button (click)="selectedCustomer=customer">Select</button>
    </tr>
```

Read and learn more about template input variables.

{@a token}

token

An opaque identifier used for efficient table lookup. In Angular, a DI token is used to find providers of dependencies in the dependency injection system.

{@a transpile}

transpile

The translation process that transforms one version of JavaScript to another version; for example, down-leveling ES2015 to the older ES5 version.

{@a file-tree}

tree

In schematics, a virtual file system represented by the `Tree` class. Schematic rules take a tree object as input, operate on them, and return a new tree object.

{@a typescript}

TypeScript

A programming language based on JavaScript that is notable for its optional typing system. TypeScript provides compile-time type checking and strong tooling support (such as code completion, refactoring, inline documentation, and intelligent search). Many code editors and IDEs support TypeScript either natively or with plug-ins.

TypeScript is the preferred language for Angular development. Read more about TypeScript at typescriptlang.org.

TypeScript configuration file

A file specifies the root files and the compiler options required to compile a TypeScript project. For more information, see TypeScript configuration.

{@a U}

{@a unidirectional-data-flow}

unidirectional data flow

A data flow model where the component tree is always checked for changes in one direction (parent to child), which prevents cycles in the change detection graph.

In practice, this means that data in Angular flows downward during change detection. A parent component can easily change values in its child components because the parent is checked first. A failure could occur, however, if a child component tries to change a value in its parent during change detection (inverting the expected data flow), because the parent component has already been rendered. In development mode, Angular throws the `ExpressionChangedAfterItHasBeenCheckedError` error if your application attempts to do this, rather than silently failing to render the new value.

To avoid this error, a lifecycle hook method that seeks to make such a change should trigger a new change detection run. The new run follows the same direction as before, but succeeds in picking up the new value.

{@a universal}

Universal

A tool for implementing server-side rendering of an Angular application. When integrated with an app, Universal generates and serves static pages on the server in response to requests from browsers. The initial static page serves as a fast-loading placeholder while the full application is being prepared for normal execution in the browser.

To learn more, see Angular Universal: server-side rendering.

{@a V}

{@a view}

view

The smallest grouping of display elements that can be created and destroyed together. Angular renders a view under the control of one or more directives.

A component class and its associated template define a view. A view is specifically represented by a `ViewRef` instance associated with a component. A view that belongs immediately to a component is called a *host view*. Views are typically collected into view hierarchies.

Properties of elements in a view can change dynamically, in response to user actions; the structure (number and order) of elements in a view can't. You can change the structure of elements by inserting, moving, or removing nested views within their view containers.

View hierarchies can be loaded and unloaded dynamically as the user navigates through the application, typically under the control of a router.

{@a ve}

View Engine

A previous compilation and rendering pipeline used by Angular. It has since been replaced by Ivy and is no longer in use. View Engine was deprecated in version 9 and removed in version 13.

{@a view-tree}

view hierarchy

A tree of related views that can be acted on as a unit. The root view is a component's *host view*. A host view can be the root of a tree of *embedded views*, collected in a *view container* (`ViewContainerRef`) attached to an anchor element in the hosting component. The view hierarchy is a key part of Angular change detection.

The view hierarchy doesn't imply a component hierarchy. Views that are embedded in the context of a particular hierarchy can be host views of other components. Those components can be in the same NgModule as the hosting component, or belong to other NgModules.

{@a W} {@a web-component}

web component

See custom element.

{@a workspace}

workspace

A collection of Angular projects (that is, applications and libraries) powered by the Angular CLI that are typically co-located in a single source-control repository (such as git).

The CLI `ng new` command creates a file system directory (the "workspace root"). In the workspace root, it also creates the workspace configuration file (`angular.json`) and, by default, an initial application project with the same name.

Commands that create or operate on applications and libraries (such as `add` and `generate`) must be executed from within a workspace folder.

For more information, see Workspace Configuration.

{@a cli-config}

{@a config}

workspace configuration

A file named `angular.json` at the root level of an Angular workspace provides workspace-wide and project-specific configuration defaults for build and development tools that are provided by or integrated with the Angular CLI.

For more information, see Workspace Configuration.

Additional project-specific configuration files are used by tools, such as `package.json` for the npm package manager, `tsconfig.json` for TypeScript transpilation, and `tslint.json` for TSLint.

For more information, see Workspace and Project File Structure.

{@a X}

{@a Y}

{@a Z} {@a zone}

zone

An execution context for a set of asynchronous tasks. Useful for debugging, profiling, and testing applications that include asynchronous operations such as event processing, promises, and calls to remote servers.

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

A zone client can take action before and after an async operation completes.

Learn more about zones in this Brian Ford video.