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` 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` 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.

> For a more detailed discussion, see Introduction to modules.

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()` 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.
>
> Learn more about decorators on the web.

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.

> For a more detailed discussion of these concepts, see Introduction to components.

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()` decorator. The decorator provides the metadata that allows other providers to be **injected** as dependencies into your class.

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.

> For a more detailed discussion, see Introduction to services and DI.

Routing

The Angular `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.

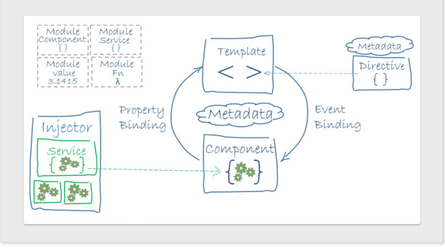
The router interprets a link URL according to your app's view navigation rules and data state. You can navigate to new views when the user clicks a button or selects from a drop box, or in response to some other stimulus from any source. The router logs activity in the browser's history, so the back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

> For a more detailed discussion, see Routing and navigation.

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.

Each of these subjects is introduced in more detail in the following pages.

- Introduction to Modules
- Introduction to Components
  - Templates and views
  - Component metadata
  - Data binding
  - Directives
  - Pipes
- Introduction to services and dependency injection

> Note that the code referenced on these pages is available as a live example / download example.

When you're familiar with these fundamental building blocks, you can explore them in more detail in the documentation. To learn about more tools and techniques that are available to help you build and deploy Angular applications, see Next steps: tools and techniques.