Architecture Overview

Angular is a framework for building client applications in HTML and either JavaScript or a language like TypeScript that compiles to JavaScript.

The framework consists of several libraries, some of them core and some optional.

You write Angular applications by composing HTML *templates* with Angularized markup, writing *component* classes to manage those templates, adding application logic in *services*, and boxing components and services in *modules*.

Then you launch the app by *bootstrapping* the *root module*. Angular takes over, presenting your application content in a browser and responding to user interactions according to the instructions you've provided.

Of course, there is more to it than this. You'll learn the details in the pages that follow. For now, focus on the big picture.

The code referenced on this page is available as a .

Modules

Angular apps are modular and Angular has its own modularity system called *NgModules*.

NgModules are a big deal. This page introduces modules; the NgModules page covers them in depth.

Every Angular app has at least one NgModule class, the *root module*, conventionally named `AppModule` .

While the *root module* may be the only module in a small application, most apps have many more *feature modules*, each a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities.

An NgModule, whether a *root* or *feature*, is a class with an `@NgModule` decorator.

Decorators are functions that modify JavaScript classes. Angular has many decorators that attach metadata to

classes so that it knows what those classes mean and how they should work. Learn more about decorators on the web.

`NgModule` is a decorator function that takes a single metadata object whose properties describe the module. The most important properties are: * `declarations` - the *view classes* that belong to this module. Angular has three kinds of view classes: components, directives, and pipes.

- `exports` - the subset of declarations that should be visible and usable in the component templates of other modules.
- `imports` - other modules whose exported classes are needed by component templates declared in *this* module.
- `providers` - creators of services that this module contributes to the global collection of services; they become accessible in all parts of the app.
- `bootstrap` - the main application view, called the *root component*, that hosts all other app views. Only the *root module* should set this `bootstrap` property.

Here's a simple root module:

The `export` of `AppComponent` is just to show how to use the `exports` array to export a component; it isn't actually necessary in this example. A root module has no reason to _export_ anything because other components don't need to _import_ the root module.

Launch an application by *bootstrapping* its root module. During development you're likely to bootstrap the `AppModule` in a `main.ts` file like this one.

NgModules vs. JavaScript modules

The NgModule — a class decorated with `@NgModule` — is a fundamental feature of Angular.

JavaScript also has its own module system for managing collections of JavaScript objects. It's completely different and unrelated to the NgModule system.

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.

Learn more about the JavaScript module system on the web.

These are two different and *complementary* module systems. Use them both to write your apps.

Angular libraries

Angular ships as a collection of JavaScript modules. You can think of them as library modules.

Each Angular library name begins with the `@angular` prefix.

You install them with the **npm** package manager and import parts of them with JavaScript `import` statements.

For example, import Angular's `Component` decorator from the `@angular/core` library like this:

You also import NgModules from Angular *libraries* using JavaScript import statements:

In the example of the simple root module above, the application module needs material from within that `BrowserModule`. To access that material, add it to the `@NgModule` metadata `imports` like this.

In this way you're using both the Angular and JavaScript module systems *together*.

It's easy to confuse the two systems because they share the common vocabulary of "imports" and "exports". Hang in there. The confusion yields to clarity with time and experience.

Learn more from the [NgModules](guide/ngmodule) page.

Components

A *component* controls a patch of screen called a *view*.

For example, the following views are controlled by components:

- 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.

{@a component-code}

For example, this `HeroListComponent` has a `heroes` property that returns an array of heroes that it acquires from a service. `HeroListComponent` also has a `selectHero()` method that sets a `selectedHero` property when the user clicks to choose a hero from that list.

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, like `ngOnInit()` declared above.

Templates

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.

A template looks like regular HTML, except for a few differences. Here is a template for our `HeroListComponent`:

Although this template uses typical HTML elements like `<h2>` and `<p>`, it also has some differences. Code like `*ngFor`, `{{hero.name}}`, `(click)`, `[hero]`, and `<hero-detail>` uses Angular's template syntax.

In the last line of the template, the `<hero-detail>` tag is a custom element that represents a new component, `HeroDetailComponent`.

The `HeroDetailComponent` is a *different* component than the `HeroListComponent` you've been reviewing. The `HeroDetailComponent` (code not shown) presents facts about a particular hero, the hero that the user selects from the list presented by the `HeroListComponent`. The `HeroDetailComponent` is a **child** of the `HeroListComponent`.

Notice how `<hero-detail>` rests comfortably among native HTML elements. Custom components mix seamlessly with native HTML in the same layouts.

Metadata

Metadata tells Angular how to process a class.

Looking back at the code for `HeroListComponent`, you can see that it's just a class. There is no evidence of a framework, no "Angular" in it at all.

In fact, `HeroListComponent` really is *just a class*. It's not a component until you *tell Angular about it*.

To tell Angular that `HeroListComponent` is a component, attach **metadata** to the class.

In TypeScript, you attach metadata by using a **decorator**. Here's some metadata for `HeroListComponent`:

Here is the `@Component` decorator, which identifies the class immediately below it as a component class.

The `@Component` decorator takes a required configuration object with the information Angular needs to create and present the component and its view.

Here are a few of the most useful `@Component` configuration options:

- `selector`: CSS selector that tells Angular to create and insert an instance of this component where it finds a `<hero-list>` tag in *parent* HTML. For example, if an app's HTML contains `<hero-list></hero-list>`, then Angular inserts an instance of the `HeroListComponent` view between those tags.
- `templateUrl`: module-relative address of this component's HTML template, shown above.
- `providers`: array of **dependency injection providers** for services that the component requires. This is one way to tell Angular that the component's constructor requires a `HeroService` so it can get the list of heroes to display.

The metadata in the `@Component` tells Angular where to get the major building blocks you specify for the component.

The template, metadata, and component together describe a view.

Apply other metadata decorators in a similar fashion to guide Angular behavior. `@Injectable`, `@Input`, and `@Output` are a few of the more popular decorators.

The architectural takeaway is that you must add metadata to your code so that Angular knows what to do.

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/pull logic by hand is tedious, error-prone, and a nightmare to read as any experienced jQuery programmer can attest.

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

As the diagram shows, there are four forms of data binding syntax. Each form has a direction — to the DOM, from the DOM, or in both directions.

The `HeroListComponent` example template has three forms:

- The `{{hero.name}}` *interpolation* displays the component's `hero.name` property value within the `<li>` element.
- The `[hero]` *property binding* passes the value of `selectedHero` from the parent `HeroListComponent` to the `hero` property of the child `HeroDetailComponent`.
- The `(click)` *event binding* calls the component's `selectHero` method when the user clicks a hero's name.

Two-way data binding is an important fourth form that combines property and event binding in a single notation, using the `ngModel` directive. Here's an example from the `HeroDetailComponent` template:

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 per 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.

Data binding is also important for communication between parent and child components.

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` decorator. A component is a *directive-with-a-template*; a `@Component` decorator is actually a `@Directive` decorator extended with template-oriented features.

While **a component is technically a directive**, components are so distinctive and central to Angular applications that this architectural overview separates components from directives.

Two *other* kinds of directives exist: *structural* and *attribute* directives.

They tend to appear within an element tag as attributes do, sometimes by name but more often as the target of an assignment or a binding.

Structural directives alter layout by adding, removing, and replacing elements in DOM.

The example template uses two built-in structural directives:

- `*ngFor` tells Angular to stamp out one `<li>` per hero in the `heroes` list.
- `*ngIf` includes the `HeroDetail` component only if a selected hero exists.

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

The `ngModel` directive, which implements two-way data binding, is an example of an attribute directive. `ngModel` modifies the behavior of an existing element (typically an `<input>`) by setting its display value property and responding to change events.

Angular has a few more directives that either alter the layout structure (for example, ngSwitch) or modify aspects of DOM elements and components (for example, ngStyle and ngClass).

Of course, you can also write your own directives. Components such as `HeroListComponent` are one kind

of custom directive.

Services

Service is a broad category encompassing any value, function, or feature that your application needs.

Almost anything can be a service. A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well.

Examples include:

- logging service
- data service
- message bus
- tax calculator
- application configuration

There is nothing specifically *Angular* about services. Angular has no definition of a service. There is no service base class, and no place to register a service.

Yet services are fundamental to any Angular application. Components are big consumers of services.

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

Here's a `HeroService` that uses a Promise to fetch heroes. The `HeroService` depends on the `Logger` service and another `BackendService` that handles the server communication grunt work.

Services are everywhere.

Component classes should be lean. They don't fetch data from the server, validate user input, or log directly to the console. They delegate such tasks to services.

A component's job is to enable the user experience and nothing more. It mediates between the view (rendered by the template) and the application logic (which often includes some notion of a *model*). A good component presents properties and methods for data binding. It delegates everything nontrivial to services.

Angular doesn't *enforce* these principles. It won't complain if you write a "kitchen sink" component with 3000 lines.

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*.

Dependency injection

Dependency injection is a way to supply a new instance of a class with the fully-formed dependencies it requires. Most dependencies are services. Angular uses dependency injection to provide new components with the services they need.

Angular can tell which services a component needs by looking at the types of its constructor parameters. For example, the constructor of your `HeroListComponent` needs a `HeroService`:

When Angular creates a component, it first asks an **injector** for the services that the component requires.

An injector maintains a container of service instances that it has previously created. If a requested service instance is not in the container, the injector makes one and adds it to the container 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. This is *dependency injection*.

The process of `HeroService` injection looks a bit like this:

If the injector doesn't have a `HeroService`, how does it know how to make one?

In brief, you must have previously registered a **provider** of the `HeroService` with the injector. A provider is something that can create or return a service, typically the service class itself.

You can register providers in modules or in components.

In general, add providers to the root module so that the same instance of a service is available everywhere.

Alternatively, register at a component level in the `providers` property of the `@Component` metadata:

Registering at a component level means you get a new instance of the service with each new instance of that component.

Points to remember about dependency injection:

- Dependency injection is wired into the Angular framework and used everywhere.
- The *injector* is the main mechanism.
    - An injector maintains a *container* of service instances that it created.
    - An injector can create a new service instance from a *provider*.
- A *provider* is a recipe for creating a service.
- Register *providers* with injectors.

Wrap up

You've learned the basics about the eight main building blocks of an Angular application:

- Modules
- Components
- Templates
- Metadata
- Data binding
- Directives
- Services
- Dependency injection

That's a foundation for everything else in an Angular application, and it's more than enough to get going. But it doesn't include everything you need to know.

Here is a brief, alphabetical list of other important Angular features and services. Most of them are covered in this documentation (or soon will be).

> **Animations**: Animate component behavior without deep knowledge of animation techniques or CSS with Angular's animation library.
>
> **Change detection**: The change detection documentation will cover how Angular decides that a component property value has changed, when to update the screen, and how it uses **zones** to intercept asynchronous activity and run its change detection strategies.
>
> **Events**: The events documentation will cover how to use components and services to raise events with mechanisms for publishing and subscribing to events.
>
> **Forms**: Support complex data entry scenarios with HTML-based validation and dirty checking.

HTTP: Communicate with a server to get data, save data, and invoke server-side actions with an HTTP client.

Lifecycle hooks: Tap into key moments in the lifetime of a component, from its creation to its destruction, by implementing the lifecycle hook interfaces.

Pipes: Use pipes in your templates to improve the user experience by transforming values for display. Consider this `currency` pipe expression:

```
price | currency:'USD':true
```

It displays a price of 42.33 as `$42.33`.

Router: Navigate from page to page within the client application and never leave the browser.

Testing: Run unit tests on your application parts as they interact with the Angular framework using the *Angular Testing Platform*.