The Ultimate ANGULAR Cheat Sheet

TABLE OF CONTENTS

Starting a New Angular Project

Before starting a new project, Node.js must be installed on your machine. Next, Angular has an official CLI tool for managing projects. It can be installed with NPM or Yarn.

```
# NPM
npm install -g @angular/cli

# Yarn
yarn global add @angular/cli
```

Afterward, we can create a new project with the following command:

```
ng new my-app
```

Angular will prompt you to configure the project. For the default settings, you can press the **Enter** or **Return** keys. During the installation process, Angular will scaffold a default project with packages for running Angular.

You can run a project with either command:

```
# Development
ng serve

# Production
ng build --prod
```

Installing a Library

Without a doubt, you will find yourself installing 3rd party libraries from other developers. Packages optimized for Angular may be installed with a special command that will install and configure a package with your project. If a package is not optimized for Angular, you have the option of installing it the traditional way.

```
# Installation + Configuration
ng add @angular/material

# Installation
npm install @angular/material
```

Creating Components

Components are the buildings blocks of an application. You can think of them as a feature for teaching browsers new HTML tags with custom behavior. Components can be created with the CLI. Typically, Angular offers a shorthand command for those who prefer to be efficient.

```
# Common
ng generate component MyComponent

# Shorthand
ng g c MyComponent
```

Angular will generate the component files in a directory of the same name. You can expect the following.

- ***.component.html** - The template of the component that gets displayed when the component is rendered.
- ***.component.css** - The CSS of a component, which is encapsulated.
- ***.component.js** - The business logic of a component to dictate its behavior.
- ***.component.spec.js** - A test file for validating the behavior and output of a component.

Along with creating the files, component classes are decorated with the `@Component` decorator and registered with the closest module. Here are some common helpful options:

Option	Example	Description
`--dry-run` (`-d`)	`ng g c MyComponent -d`	Does not output the result. Useful for keeping your command line clean.
`--export`	`ng g c MyComponent --export`	Exports the component in the module's `exports` option.
`--force` (`f`)	`ng g c MyComponent --f`	Forces a component to be created even if it already exists. Useful for overwriting files.
`--help`	`ng g c --help`	Outputs a complete list of options for a given command.
`--prefix` (`-p`)	`ng g c MyComponent -p=base`	Custom prefix for a component's HTML selector
`--skip-tests`	`ng g c MyComponent --skip-tests`	Skips creating the ****.spec.ts** file.
`--style`	`ng g c MyComponent --style=scss`	A file extension or preproccessor for the style files. Can be set to `'none'` to skip generating a style file.

Lifecycle Hooks

Components emit events during and after initialization. Angular allows us to hook into these events by defining a set of methods in a component's class. You can dive deeper into hooks here.

Here's a quick rundown on the lifecycle hooks available:

- `ngOnChanges`: Runs after an input/output binding has been changed.
- `ngOnInit`: Runs after a component has been initialized. Input bindings are ready.
- `ngDoCheck`: Allows developers to perform custom actions during change detection.
- `ngAfterContentInit`: Runs after the content of a component has been initialized.
- `ngAfterContentChecked`: Runs after every check of a component's content.
- `ngAfterViewInit`: Runs after the view of a component has been initialized.
- `ngAfterViewChecked`: Runs after every check of a component's view.
- `ngOnDestroy`: Runs before a component is destroyed.

Services

Services are objects for outsourcing logic and data that can be injected into our components. They're handy for reusing code in multiple components. For medium-sized apps, they can serve as an alternative to state management libraries.

We can create services with commands:

```
# Common
ng generate service MyService

# Shorthand
ng g s MyService
```

Services are not standalone. Typically, they're injected into other areas of our app. Most commonly in components. There are two steps for injecting a service. First, we must add the `@Injectable()` decorator.

```
import { Injectable } from '@angular/core';

@Injectable()
export class MyService {
  constructor() { }
}
```

Secondly, we must tell Angular *where* to inject this class. There are three options at our disposal.

1. Injectable Decorator

This option is the most common route. It allows the service to be injectable anywhere in our app.

```
@Injectable({
  providedIn: 'root'
})
```

2. Module

This option allows a service to be injectable in classes that are imported in the same module.

```
@NgModule({
  declarations: [],
  imports: [],
  providers: [MyService}],
  bootstrap: []
})
```

3. Component Class

This option allows a service to be injectable in a single component class.

```
@Component({
  providers: [MyService]
})
```

Once you've got those two steps settled, a service can be injected into the `constructor()` function of a class:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-example',
  template: '<p>Hello World</p>',
  styleUrls: ['./example.component.css']
})
export class ExampleComponent {
  constructor(private myService: MyService) { }
}
```

Modules

Angular enhances JavaScript's modularity with its own module system. Classes decorated with the `@NgModule()` decorator can register components, services, directives, and pipes.

The following options can be added to a module:

- `declarations` - List of components, directives, and pipes that belong to this module.
- `imports` - List of modules to import into this module. Everything from the imported modules is available to declarations of this module.
- `exports` - List of components, directives, and pipes visible to modules that import this module.
- `providers` - List of dependency injection providers visible both to the contents of this module and to importers of this module.
- `bootstrap` - List of components to bootstrap when this module is bootstrapped.

Here's an example of a module called `AppModule`.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Angular Directives

Directives are custom attributes that can be applied to elements and components to modify their behavior. There are two types of directives: **attribute directives** and **structural directives**.

Attribute Directives

An attribute directive is a directive that changes the appearance or behavior of an element, component, or another directive. Angular exports the following attribute directives:

`NgClass`

Adds and removes a set of CSS classes.

```
<!-- toggle the "special" class on/off with a property -->
<div [ngClass]="isSpecial ? 'special' : ''">This div is special</div>
```

`NgStyle`

Adds and removes a set of HTML styles.

```
<div [ngStyle]="{
  'font-weight': 2 + 2 === 4 ? 'bold'   : 'normal',
}">
  This div is initially bold.
</div>
```

`NgModel`

Adds two-way data binding to an HTML form element. Firstly, this directive requires the `FormsModule` to be added to the `@NgModule()` directive.

```
import { FormsModule } from '@angular/forms'; // <--- JavaScript import from Angular
/* . . . */
@NgModule({
  /* . . . */
  imports: [
    BrowserModule,
    FormsModule // <--- import into the NgModule
  ],
  /* . . . */
})
export class AppModule { }
```

Secondly, we can bind the `[(ngModel)]` directive on an HTML `<form>` element and set it equal to the property.

```
<label for="example-ngModel">[(ngModel)]:</label>
<input [(ngModel)]="currentItem.name" id="example-ngModel">
```

The `NgModel` directive has more customizable options that can be [found here](https://angular.io/guide/built-in-directives#displaying-and-updating-properties-with-ngmodel).

Structural Directives

Structural directives are directives that change the DOM layout by adding and removing DOM elements. Here are the most common structural directives in Angular:

`NgIf`

A directive that will conditionally create or remove elements from the template. If the value of the `NgIf` directive evaluates to `false`, Angular removes the element.

```
<p *ngIf="isActive">Hello World!</p>
```

`NgFor`

Loops through an element in a list/array.

```
<div *ngFor="let item of items">{{item.name}}</div>
```

`NgSwitch`

An alternative directive for conditionally rendering elements. This directive acts very similarly to the JavaScript `switch` statement. There are three directives at our disposal:

- `NgSwitch` — A structural directive that should be assigned the value that should be matched against a series of conditions.
- `NgSwitchCase` — A structural directive that stores a possible value that will be matched against the `NgSwitch` directive.
- `NgSwitchDefault` — A structural directive that executes when the expression doesn't match with any defined values.

```
<ul [ngSwitch]="food">
  <li *ngSwitchCase="'Burger'">Burger</li>
  <li *ngSwitchCase="'Pizza'">Pizza</li>
  <li *ngSwitchCase="'Spaghetti'">Spaghetti</li>
  <li *ngSwitchDefault>French Fries</li>
</ul>
```

Custom Directives

We're not limited to directives defined by Angular. We can create custom directives with the following command:

```
# Common
ng generate directive MyDirective

# Shorthand
ng g d MyDirective
```

To identify directives, classes are decorated with the `@Directive()` decorator. Here's what a common directive would look like:

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appMyDirective]'
})
export class appMyDirective {
  constructor(private elRef: ElementRef) {
    eleRef.nativeElement.style.background = 'red';
```

```
  }
}
```

Pipes

Pipes are known for transforming content but not directly affecting data. They're mainly utilized in templates like so:

```
{{ 'Hello world' | uppercase }}
```

Angular has a few pipes built-in.

`DatePipe`

Formats a date value according to locale rules.

```
{{ value_expression | date 'short' }}
```

`UpperCasePipe`

Transforms text to all upper case.

```
{{ 'Hello world' | uppercase }}
```

`LowerCasePipe`

Transforms text to all lower case.

```
{{ 'Hello World' | lowercase }}
```

`CurrencyPipe`

Transforms a number to a currency string, formatted according to locale rules.

```
{{ 1.3495 | currency:'CAD' }}
```

`DecimalPipe`

Transforms a number into a string with a decimal point, formatted according to locale rules.

```
{{ 3.14159265359 | number }}
```

`PercentPipe`

Transforms a number to a percentage string, formatted according to locale rules.-

```
{{ 0.259 | percent }}
```

Decorators

Angular exports dozens of decorators that can be applied to classes and fields. These are some of the most common decorators you'll come across.

Decorator	Example	Description
`@Input()`	`@Input() myProperty`	A property can be updated through property binding.
`@Output()`	`@Output() myEvent = new EventEmitter();`	A property that can fire events and can be subscribed to with event binding on a component.
`@HostBinding()`	`@HostBinding('class.valid') isValid`	Binds a host element property (here, the CSS class valid) to a directive/component property (isValid).
`@HostListener()`	`@HostListener('click', ['$event']) onClick(e) {...}`	A directive for subscribing to an event on a host element, such as a `click` event, and run a method when that event is emitted. You can optionally accept the `$event` object.
`@ContentChild()`	`@ContentChild(myPredicate) myChildComponent;`	Binds the first result of the component content query (`myPredicate`) to a property (`myChildComponent`) of the class.
`@ContentChildren()`	`@ContentChildren(myPredicate) myChildComponents;`	Binds the results of the component content query (`myPredicate`) to a property (`myChildComponents`) of the class.
`@ViewChild()`	`@ViewChild(myPredicate) myChildComponent;`	Binds the first result of the component view query (`myPredicate`) to a property (`myChildComponent`) of the class. Not available for directives.
`@ViewChildren()`	`@ViewChildren(myPredicate) myChildComponents;`	Binds the results of the component view query (`myPredicate`) to a property (`myChildComponents`) of the class. Not available for directives.

Useful Links

- Angular Documentation
- Angular Devtools

- Angular API Reference
- Angular Blog
- Angular Routing
- Angular Forms
- ZTM Angular Bootcamp

Back To Top