Angular-Cheat-Sheet

Angular Cheat Sheet: Zero To Mastery

Table of Contents

1. Starting a New Project
2. Installing a Library
3. Creating Components
4. Lifecycle Hooks
5. Services
6. Modules
7. Angular Directives
8. Binding in Angular
9. Pipes
10. Decorators
11. Angular Routing
12. Angular HTTP Client
13. Angular Testing
14. Useful Links

Starting a New Project

Before starting a new project, ensure that Node.js is installed on your machine. Angular provides an official CLI tool for managing projects, which can be installed via NPM or Yarn.

Installation:

NPM:

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

Yarn:

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

Creating a New Project:

```
ng new my-app
```

Angular will prompt you to configure the project. For default settings, press Enter/Return keys. During installation, Angular will scaffold a default project with necessary packages.

To run the project:

```
# Development
ng serve

# Production
ng build --prod
```

Installing a Library

You may need to install 3rd-party libraries for your project. Angular provides a special command to install and configure Angular-optimized packages.

Installation + Configuration:

```
ng add @angular/material
```

Installation:

```
npm install @angular/material
```

Creating Components

Components are the building blocks of an Angular application. They can be created using Angular CLI.

Common:

```
ng generate component MyComponent
```

Shorthand:

```
ng g c MyComponent
```

Angular will generate necessary files for the component in a directory with the same name.

Lifecycle Hooks

Angular components emit events during and after initialization. Lifecycle hooks allow developers to hook into these events.

Hooks:

- ngOnChanges
- ngOnInit
- ngDoCheck

- ngAfterContentInit
- ngAfterContentChecked
- ngAfterViewInit
- ngAfterViewChecked
- ngOnDestroy

Services

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

Creating a Service:

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'
})
```

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.

Example 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 modify the behavior of elements and components in Angular applications. There are two types: 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

- NgStyle
- NgModel

Detailed Explaination

- **NgClass**: Adds and removes a set of CSS classes.

```
<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 initial
```

- **NgModel**: Adds two-way data binding to an HTML form element. (Requires FormsModule to be imported into the NgModule)

```
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [FormsModule]
})
```

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

Structural Directives:

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

- NgIf
- NgFor
- NgSwitch

Detailed Explaination

- **NgIf**: Conditionally creates or removes elements from the template.

```
<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**: Conditionally renders elements using a switch-like syntax.

```
<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

Directives in Angular allow you to create custom HTML elements and attributes. They can be used to extend the behavior of existing DOM elements or create entirely new ones.

Creating a Directive:

```
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';
  }
}
```

Binding in Angular

In Angular, binding refers to the process of establishing communication between the component class (the TypeScript code) and the template (the HTML code). Bindings allow you to pass data from the component class to the template, listen to user events in the template and react accordingly, and even update data dynamically in both directions.

There are several types of bindings in Angular:

1. Property Binding:

Property binding allows you to set the value of an HTML element property dynamically based on a value in the component class. It's denoted by square brackets `[]`. For example, you can bind the `src` property of an `img` tag to a variable in the component class to dynamically change the image source.

```
<img [src]="imageUrl">
```

2. Event Binding:

Event binding allows you to listen to events raised by HTML elements, such as button clicks, and execute a method in response. It's denoted by parentheses `()`. For example, you can bind the `click` event of a button to a method in the component class to perform some action when the button is clicked.

```
<button (click)="onClick()">Click Me</button>
```

3. Two-Way Binding:

Two-way binding allows you to establish a synchronization between a property in the component class and an input field or other form element in the template. It combines property binding and event binding using the `[(ngModel)]` directive. This means changes in the input field update the component property, and changes in the component property update the input field.

```
<input [(ngModel)]="name">
```

4. One-Way Binding:

One-way binding allows data to flow in one direction only, either from the component class to the template (property binding) or from the template to the component class (event binding).

- **Property Binding (One-Way)**: Allows you to set a property of an HTML element based on a value in the component class.

  Example:

  ```
  <p [innerText]="message"></p>
  ```

- **Event Binding (One-Way)**: Allows you to listen to events raised by HTML elements and execute a method in response.

  Example:

  ```
  <button (click)="onClick()">Click Me</button>
  ```

5. Attribute Binding:

Attribute binding allows you to set HTML attributes of an element dynamically based on values in the component class. It's similar to property binding but works with HTML attributes instead.

```
<button [attr.disabled]="isDisabled ? true : null">Submit</button>
```

6. Class and Style Binding:

Class and style binding allow you to add or remove CSS classes and apply inline styles dynamically based on conditions in the component class.

```
<div [class.error]="isError">Error Message</div>
<div [style.color]="isError ? 'red' : 'black'">Error Message</div>
```

These binding mechanisms make Angular templates highly dynamic and responsive to changes in the component class. They form the foundation of building interactive and data-driven web applications with Angular.

Pipes

Pipes transform content in templates without directly affecting data. Angular provides several built-in pipes.

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

Angular has a few pipes built-in.

Built-in Pipes:

- **DatePipe**: Formats a date value according to locale rules.
- **UpperCasePipe**: Transforms text to all uppercase.
- **LowerCasePipe**: Transforms text to all lowercase.
- **CurrencyPipe**: Transforms a number to a currency string, formatted according to locale rules.
- **DecimalPipe**: Transforms a number into a string with a decimal point, formatted according to locale rules.
- **PercentPipe**: Transforms a number to a percentage string, formatted according to locale rules.

Decorators

Angular provides decorators that can be applied to classes and fields for various purposes.

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)
@HostListener()	@HostListener('click', ['$event']) onClick(e)...	Subscribes to an event on a host element, such as a click event, and runs a method u

object.

Decorator	Example	Description
@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.

Angular Routing

Routing in Angular allows you to navigate between different components and views in your application without reloading the entire page. It helps in creating Single Page Applications (SPAs) where navigation occurs without page refresh.

Setting Up Routing:

1. Define routes in the AppRoutingModule.
2. Import RouterModule and Routes from @angular/router.
3. Configure routes with path and component.
4. Add in the template where the component will be rendered.

Example:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home.component';
import { AboutComponent } from './about.component';

const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
```

```
})
export class AppRoutingModule { }
```

Angular Forms

Angular provides powerful tools for handling forms, including template-driven forms and reactive forms.

Template-driven Forms:

Template-driven forms are easy to use and suitable for simple forms with less logic.

Reactive Forms:

Reactive forms provide a more flexible and scalable approach to handling forms. They are more suitable for complex forms with dynamic behavior.

Angular HTTP Client

Angular provides a built-in HTTP client module for making HTTP requests to servers.

Usage:

1. Import the HttpClientModule.
2. Inject the HttpClient service into your component or service.
3. Use HttpClient methods like get(), post(), put(), delete(), etc., to make HTTP requests.

Example:

```
import { HttpClient } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class DataService {
  constructor(private http: HttpClient) { }

  getData() {
    return this.http.get('/api/data');
  }
}
```

Angular Testing

Angular provides tools and utilities for testing your application's components, services, and other features.

Types of Testing:

1. **Unit Testing:** Testing individual components or services in isolation.
2. **Integration Testing:** Testing how components or services work together.
3. **End-to-End Testing:** Testing the entire application flow from start to finish.

Tools:

- Jasmine: A behavior-driven development framework for testing JavaScript code.
- Karma: A test runner for executing unit tests.
- Protractor: An end-to-end testing framework for Angular applications.

Back To Top

Useful Links

- Angular Documentation
- Angular Devtools
- Angular API Reference
- Angular Blog
- Angular Routing
- Angular Forms

Let me know if you need further modifications or any mistakes.