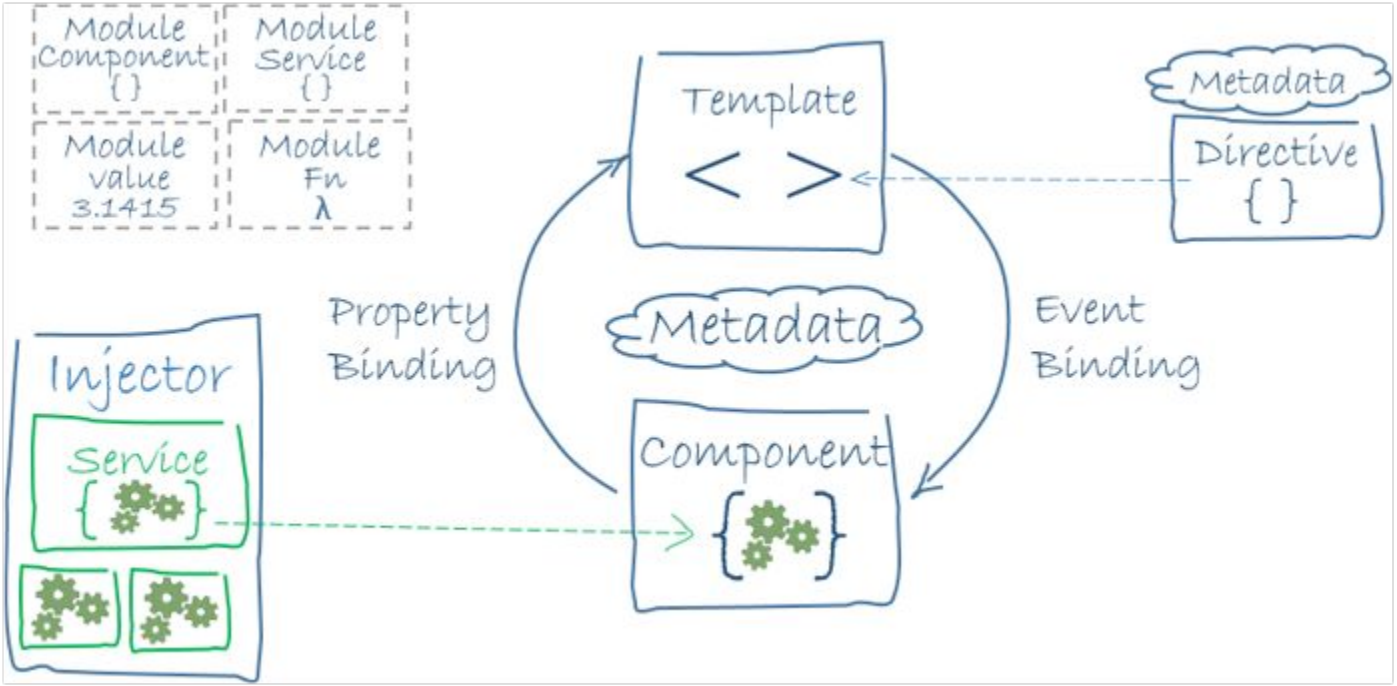
Angular.io

-Amuthan Arulraj

1

Architecture


Module
Component
{ }
Module
Service
{ }
Module
value
3.1415
Module
Fn
λ
Template
< >
Metadata
Directive
{ }
Property
Binding
Metadata
Event
Binding
Injector
Service
Component

NgModules

- Angular apps are modular and Angular has its own modularity system called NgModules.
- It can contain components, service providers, and other code files whose scope is defined by the containing NgModule.
- It can import functionality that is exported from other NgModules, and export selected functionality for use by other NgModules.
- Every Angular app has at least one NgModule class, the root module.

@NgModule

- **declarations** - The components, directives, and pipes that belong to this NgModule.
- **exports** - The subset of declarations that should be visible and usable in the component templates of other NgModules.
- **imports** - Other modules whose exported classes are needed by component templates declared in this NgModule.
- **providers** - Creators of services that this NgModule 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, which hosts all other app views. Only the root NgModule should set this bootstrap property.

Components

- A component controls a patch of screen called a view.
- A component's job is to enable the user experience and nothing more.
- It should present properties and methods for data binding, in order to mediate between the view (rendered by the template) and the application logic (which often includes some notion of a model).
- By separating a component's view-related functionality from other kinds of processing, you can make your component classes lean and efficient.

@Component

- **selector** - A CSS selector that tells Angular to create and insert an instance of this component wherever it finds the corresponding tag in template HTML.
- **templateUrl** - The module-relative address of this component's HTML template.
- **template** - The HTML template inline.
- **providers** - An array of dependency injection providers for services that the component requires.

Service

- A service is a class with a narrow, well-defined purpose. It should do something specific and do it well.
- A component should not need to define things like how to fetch data from the server, validate user input, or log directly to the console. Instead, it can delegate such tasks to services.
- By defining these kind of processing task in an injectable service class, you make it available to any component.

Dependency Injection

- Dependency Injection (DI) is a way to create objects that depend upon other objects. A Dependency Injection system supplies the dependent objects (called the dependencies) when it creates an instance of an object.
- Dependency injection (often called DI) is wired into the Angular framework.
- Angular creates an application-wide injector for you during the bootstrap process.
- The injector maintains a container of dependency instances that it has already created, and reuses them if possible.
- For any dependency you need in your app, you must register a provider with the app's injector, so that the injector can use it to create new instances.

When Angular creates a new instance of a component class, it determines which services or other dependencies that component needs by looking at the types of its constructor parameters. It first checks if the injector already has any existing instances of that service.

If a requested service instance does not yet exist, the injector makes one using the registered provider, and adds it to the injector 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.

2

Data Binding

Interpolation

- Angular evaluates all expressions in double curly braces, converts the expression results to strings, and links them with neighboring literal strings. Finally, it assigns this composite interpolated result to an element or directive property.
- It is a type of one-way binding.
- The expression context is typically the component instance.
- {{ 1 + 1 + getVal() }}

Interpolation

- No visible side effects
- Quick execution
- Simplicity
- Idempotence

One-way binding (Interpolation)

One-way from data source to view target. Target can be a DOM property.

{{expression}}

[target]="expression"

One-way binding (Interpolation)

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

<app-hero-detail [hero]="currentHero"></app-hero-detail>

<div [ngClass]="{'special': isSpecial}"></div>

<div [class.special]="isSpecial">Special</div>

<button [style.color]="isSpecial ? 'red' : 'green'">

<button [attr.aria-label]="help">help</button>
```

One-way binding (Event)

One-way from view target to data source

(target)="statement"

One-way binding (Event)

```
<button (click)="onSave()">Save</button>
<app-hero-detail
(deleteRequest)="deleteHero()"></app-hero-detail>
<div (myClick)="clicked=$event" clickable>click me</div>
<input type="text" #box
(keyup.enter)="addHero(box.value); box.value=""
placeholder="hero name">
```

Two-way binding

Banana in a box [()]

[(target)]="expression"

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

HTML attribute vs. DOM property

- Attributes are defined by HTML. Properties are defined by the DOM (Document Object Model).
- Attributes initialize DOM properties and then they are done. Property values can change; attribute values can't.
- The HTML attribute and the DOM property are not the same thing, even when they have the same name.

3

Component Interaction

Input binding

Pass data from parent to child with input binding.

```
@Input() hero: Hero;
```

```
<app-hero-child [hero]="hero">
```

Setter

Intercept input property changes with a setter.

```
@Input()
 set name(name: string) {
   this._name = name.toUppercase();
 }
 get name(): string { return this._name; }
<app-name-child [name]="name"></app-name-child>
```

Event

Parent listens for child event.

```
//Child
@Output() onVoted = new EventEmitter<boolean>();
  vote(agreed: boolean) {
    this.onVoted.emit(agreed);
  }
//Parent
  <app-voter *ngFor="let voter of voters"
      [name]="voter" (onVoted)="onVoted($event)">
```

Local Variable

Parent interacts with child via local variable.

```
<div class="seconds">{{timer.seconds}}</div>
<app-countdown-timer
#timer></app-countdown-timer>
```

4

Directives

Directives

There are three kinds of directives in Angular:

- **Components** - directives with a template.
- **Structural directives** - change the DOM layout by adding and removing DOM elements.
- **Attribute directives** - change the appearance or behavior of an element, component, or another directive.
- NgIf (upperCamelCase) refers to the directive class; ngIf (lowerCamelCase) refers to the directive's attribute name.
- The asterisk is "syntactic sugar" for something a bit more complicated.
- Angular translates the *ngIf attribute into a <ng-template> element, wrapped around the host element

Attribute Directives

An Attribute directive changes the appearance or behavior of a DOM element.

```
<p appHighlight>Highlight me!</p>
```

Attribute Directives

```
import { Directive, ElementRef } from '@angular/core';
@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
    constructor(el: ElementRef) {
      el.nativeElement.style.backgroundColor = 'yellow';
    }
}
```

Structural Directives

Structural directives are responsible for HTML layout. They shape or reshape the DOM's structure, typically by adding, removing, or manipulating elements.

```
<div *ngIf="hero" class="name">{{hero.name}}</div>
<ul>
  <li *ngFor="let hero of heroes">{{hero.name}}</li>
</ul>
```

Structural Directives

```
<div [ngSwitch]="hero?.emotion">
  <app-happy-hero    *ngSwitchCase="'happy'"
[hero]="hero"></app-happy-hero>
  <app-sad-hero      *ngSwitchCase="'sad'"
[hero]="hero"></app-sad-hero>
  <app-confused-hero *ngSwitchCase="'app-confused'"
[hero]="hero"></app-confused-hero>
  <app-unknown-hero  *ngSwitchDefault
[hero]="hero"></app-unknown-hero>
</div>
```

5

Pipe

Pipes

- A way to write display-value transformations that you can declare in your HTML.
- A pipe takes in data as input and transforms it to a desired output.
- For example, in most use cases, users prefer to see a date in a simple format like April 15, 1988 rather than the raw string format Fri Apr 15 1988 00:00:00 GMT-0700 (Pacific Daylight Time).

Pipes

```
<p>The hero's birthday is {{ birthday | date }}</p>

<p>The hero's birthday is {{ birthday | date:"MM/dd/yy" }} </p>

{{  birthday | date:'fullDate' | uppercase}}

<div *ngFor="let hero of (heroes | flyingHeroes)">

  {{hero.name}}

</div>
```

Custom Pipes

```
//Custom Pipe
@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
  transform(value: number, exponent: string): number {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
}
```

Built-in pipes

- DatePipe
- UpperCasePipe
- LowerCasePipe
- CurrencyPipe
- PercentPipe

6

Lifecycle Hooks

Lifecycle Hooks

Directive and component instances have a lifecycle as Angular creates, updates, and destroys them. Developers can tap into key moments in that lifecycle by implementing one or more of the lifecycle hook interfaces in the Angular core library.

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().

Lifecycle Hooks

- **ngOnChanges()** - Respond when Angular (re)sets data-bound input properties.
- **ngOnInit()** - Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties. Called once, after the first ngOnChanges().
- **ngOnDestroy()** - Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks.

References

- https://angular.io/guide
- https://www.typescriptlang.org/docs/handbook