Contents

[Components 3](#_Toc19678062)

[- Selector 3](#_Toc19678063)

[- Providers 3](#_Toc19678064)

[- Pipes 3](#_Toc19678065)

[- Directives 3](#_Toc19678066)

[o Components 3](#_Toc19678067)

[o Structural 3](#_Toc19678068)

[o Attribute directives 3](#_Toc19678069)

[Template 3](#_Toc19678070)

[Directive 3](#_Toc19678071)

[Binding 4](#_Toc19678072)

[- Event binding 4](#_Toc19678073)

[- Property binding 4](#_Toc19678074)

[- Two-way data binding 4](#_Toc19678075)

[Service 4](#_Toc19678076)

[NgModules 4](#_Toc19678077)

[o Declarations 4](#_Toc19678078)

[o providers 4](#_Toc19678079)

[Dependency injection (DI) 4](#_Toc19678080)

[Template expressions 5](#_Toc19678081)

[Template statements 5](#_Toc19678082)

[@Input() 5](#_Toc19678083)

[@Output() 5](#_Toc19678084)

[Safe navigation operator ( ? ) 5](#_Toc19678085)

[Lifecycle sequence 5](#_Toc19678086)

[OnInit() 5](#_Toc19678087)

[OnChanges() 5](#_Toc19678088)

[AfterView 5](#_Toc19678089)

[AfterContent 5](#_Toc19678090)

[Component Interaction 6](#_Toc19678091)

[ComponentFactoryResolver 6](#_Toc19678092)

[Forms 6](#_Toc19678093)

[Reactive forms 6](#_Toc19678094)

[Template-driven forms 6](#_Toc19678095)

[Data flow in forms 7](#_Toc19678096)

[Reactive Forms 7](#_Toc19678097)

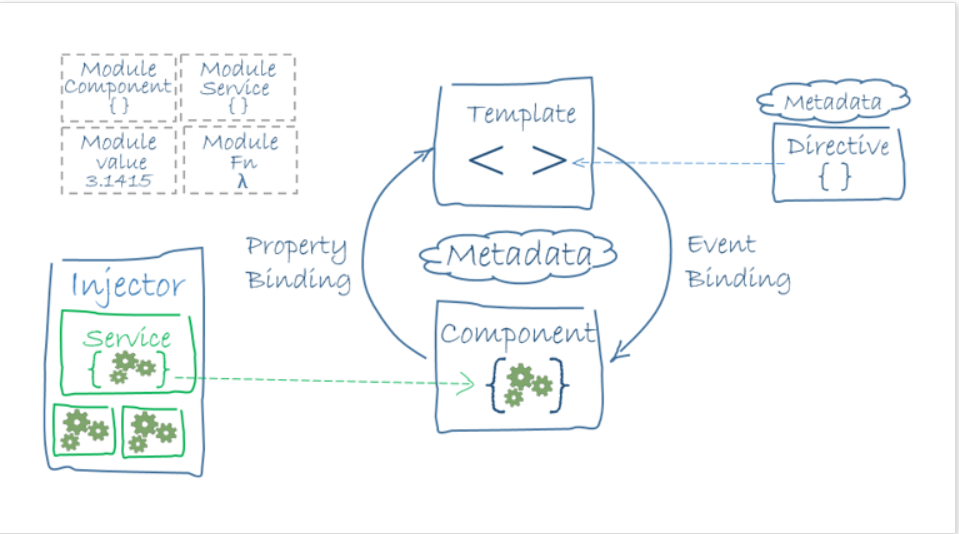
[Form Validation 7](#_Toc19678098)

[Template-driven validation 7](#_Toc19678099)

[Reactive form validation 7](#_Toc19678100)

[Built-in validators 8](#_Toc19678101)

[Custom validators 8](#_Toc19678102)



# Components

Component defines a class that contains application data and logic, and is associated with an HTML template that defines a view.

* Selector: tells Angular to create and insert an instance of this component wherever it finds the corresponding tag in template HTML.
* Providers: A provider is an object that tells an injector how to obtain or create a dependency. An array of providers for services that the component requires.
* Pipes: to transform data before it is displayed
* Directives: to apply app logic to what gets displayed.
  + Components: directives with a template.
  + Structural: change the DOM layout by adding and removing DOM elements (NgFor and NgIf.)
  + Attribute directives: change the appearance or behavior of an element, component, or another directive (NgStyle)

# Template

A template combines HTML with Angular markup that can modify HTML elements before they are displayed.

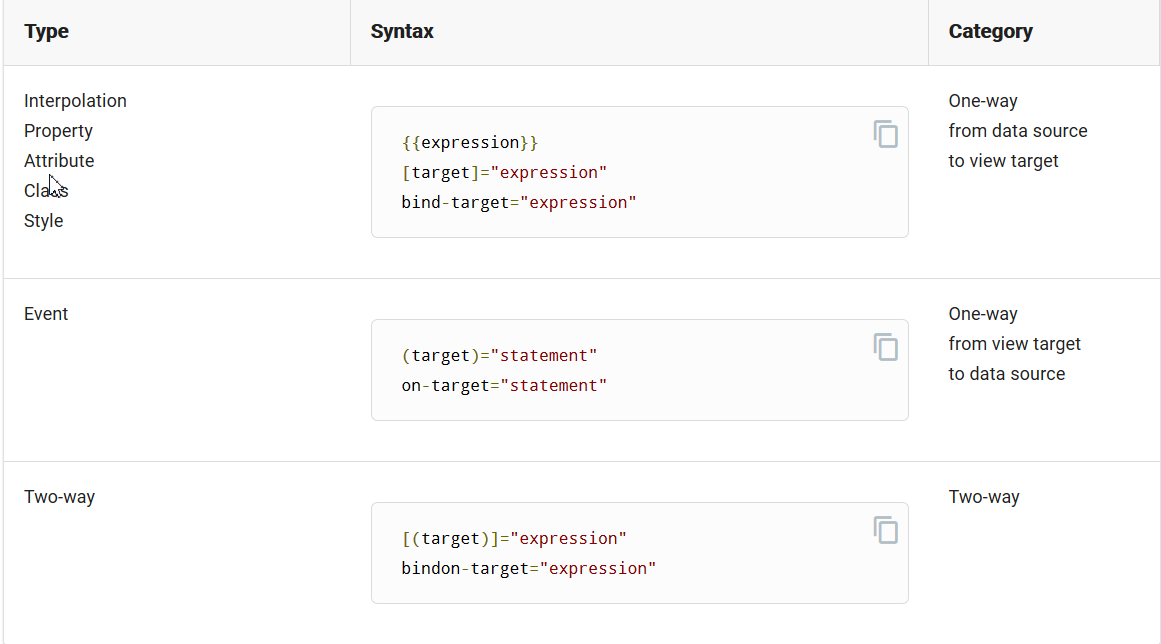
# Directive

Directives provide program logic

# Binding

Binding markup connects our application data and the DOM. 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
* Two-way data binding (used mainly in template-driven forms) combines property and event binding in a single notation.
  + Sets a specific element property.
  + Listens for an element change event.



# Service

For data or logic that isn't associated with a specific view, and that we want to share across components

* Fetching data from the server, validating user input.

# NgModules

* are containers for a cohesive block of code dedicated to an application domain
* can contain components, service providers, and other code files
  + Declarations: The components, directives, and pipes that belong to this NgModule.
  + providers: The same instance of a service is available to all components in that NgModule.

# Dependency injection (DI)

* DI is wired into the Angular framework and used everywhere to provide new components with the services or other things they need
* An injector creates dependencies, and maintains a container of dependency instances that it reuses if possible

# Template expressions

A template expression produces a value and appears within the double curly braces, {{ }}. Angular executes the expression and assigns it to a property of a binding target.

* Simplicity
* Quick execution
* No visible side effects

# Template statements

A template statement responds to an event raised by a binding target such as an element, component, or directive. A template statement has a side effect.

# @Input()

@Input() decorator in a child component or directive to let Angular know that a property in that component can receive its value from its parent component.

# @Output()

Use the @Output() decorator in the child component or directive to allow data to flow from the child out to the parent.

# Safe navigation operator ( ? )

*{{item?.name}}* If item is null, the view still renders but the displayed value is blank

# Lifecycle sequence

## OnInit()

Use ngOnInit() for two main reasons:

* To perform complex initializations shortly after construction
* To set up the component after Angular sets the input properties.

## OnChanges()

Angular calls its ngOnChanges() method whenever it detects changes to input properties of the component

## AfterView

The AfterView sample explores the AfterViewInit() and AfterViewChecked() hooks that Angular calls after it creates a component's child views

## AfterContent

The AfterContent sample explores the AfterContentInit() and AfterContentChecked() hooks that Angular calls after Angular projects external content into the component

# Component Interaction

* Pass data from parent to child with input binding
* Parent listens for child event
  + The child component exposes an EventEmitter property with which it emits events when something happens. The parent binds to that event property and reacts to those events
* Intercept input property changes with a setter
  + Use an input property setter to intercept and act upon a value from the parent.
* Intercept input property changes with ngOnChanges()
* Parent calls an @ViewChild()
* Parent and children communicate via a service

# ComponentFactoryResolver

use ComponentFactoryResolver to add components dynamically.

A simple registry that maps Components to generated ComponentFactory classes that can be used to create instances of components

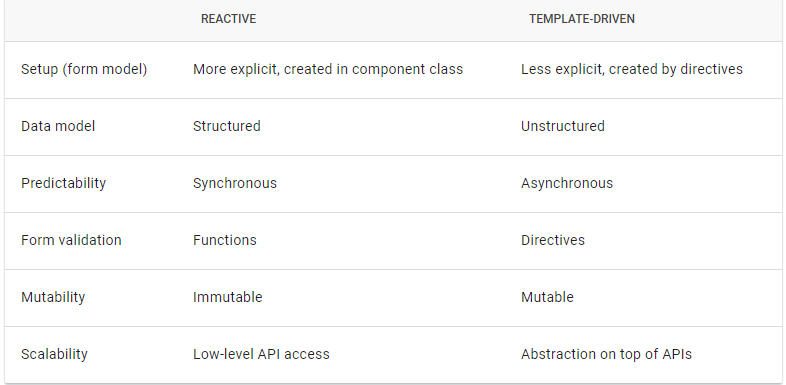
# Forms

Reactive forms: are more robust. They're more scalable, reusable, and testable.

* The reactive form directive (in this case, FormControlDirective) then links the existing FormControl instance to a specific form element in the view using a value accessor (ControlValueAccessor instance).
* Define custom validators as functions that receive a control to validate.

Template-driven forms: are useful for adding a simple form to an app.

* The template-driven form directive NgModel is responsible for creating and managing the FormControl instance for a given form element. It's less explicit, but you no longer have direct control over the form model
* Are tied to template directives, and must provide custom validator directives that wrap validation functions.



# Data flow in forms

In reactive forms each form

Element in the view is directly linked to a form model. Updates from the view to the model and from the model to the view are synchronous (The FormControl instance emits the new value through the valueChanges observable. Any subscribers to the valueChanges observable receive the new value.) <https://angular.io/guide/forms-overview#data-flow-in-reactive-forms>

Data flow in template-driven forms

Each form element is linked to a directive that manages the form model internally.

# Reactive Forms

There are two ways to update the model value:

* Use the setValue() method to set a new value for an individual control. It replaces the entire value for the control.
* Use the patchValue() method to replace any properties defined in the object that have changed in the form model.

*this.user = this.userService.loadUser().pipe(*

*tap(user => this.form.patchValue(user))*

*);*

<https://angular.io/guide/reactive-forms#dynamic-controls-using-form-arrays>

# Form Validation

## Template-driven validation

To add validation to a template-driven form, you add the same validation attributes as you would with [native HTML form validation](https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5/Constraint_validation). Angular uses directives to match these attributes with validator functions in the framework

## Reactive form validation

There are two types of validator functions:

* **Sync validators**: functions that take a control instance and immediately return either a set of validation errors or null. You can pass these in as the second argument when you instantiate a FormControl.
* **Async validators**: functions that take a control instance and return a Promise or Observable that later emits a set of validation errors or null. You can pass these in as the third argument when you instantiate a FormControl.
  + <https://alligator.io/angular/async-validators/>
* *Note: for performance reasons, Angular only runs async validators if all sync validators pass. Each must complete before errors are set*

## Built-in validators

We can choose to write your own validator functions, or you can use some of Angular's built-in validators.

## Custom validators

Since the built-in validators won't always match the exact use case of your application, sometimes you'll want to create a custom validator.

<https://angular.io/guide/form-validation#adding-to-reactive-forms>

<https://angular.io/guide/form-validation#control-status-css-classes>