COMPONENT BASICS

The primary building block

WHAT IS A COMPONENT?

A component controls a patch of screen called a view. For example, individual components define and control each of the following views from the Tutorial:

In Angular, components are made up by two parts:

- A model, represented by a class, to hold values, methods and lifecycle hooks.
- A view, represented as decorated metadata on the class, to hold template and related things.

The model sends values to the view. The View sends events to the Model.

When the Model is updated, the view is rerendered.

The @Component decorator identifies the class immediately below it as a component class, and specifies its metadata.

```
@Component({
  selector:    'container',
  templateUrl: './some-file.html',
  providers:  [ DataService ]
})
```

In the previous example we saw some of the most common options to the @Component decorater includes

- selector: A CSS selector that tells angular to inject a component where it finds this selector.
- templateURL: A path to a template file.
- providers: An array of service providers.

TEMPLATE SYNTAX

A template looks like regular HTML, except that it also contains angular template syntax.

- We can use data bindings to map data from our Model to our View.
- We can use **directives** to apply logic to the template.

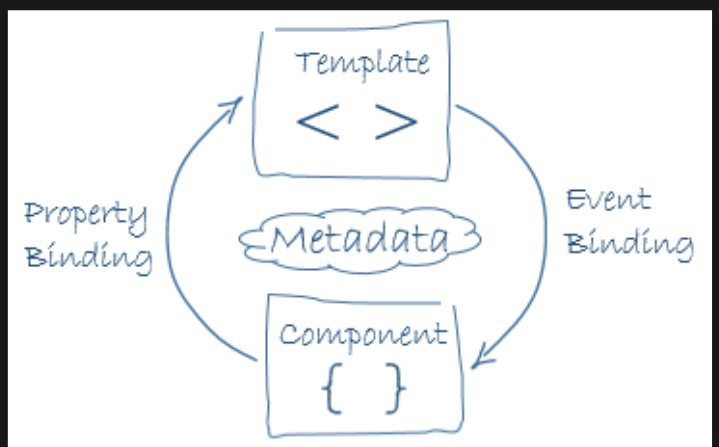
DATA BINDINGS

There are two types of data bindings.

- Event binding lets your app respond to user input.
- Property binding lets you interpolate values

Angular supports two way bindings, to connect our View and Model.

In two-way binding, a data property value flows to the View from the Component. When a change is triggered, the event flows back to the Component.



Property binding is commonly seen when we interpolate values. We bind values from our Model, to our View.

```
<div>
    {{user.name}}
</div>
```

Event binding lets us bind to DOM events, and trigger functions in our model.

```
<button (click)="sayHi(user.name)">
```

This would trigger a function sayHi in our Model. Notice how we use property values when we trigger the function.

DIRECTIVES

A **directive** transforms the DOM according to the instructions given by directive. We have already seen the most common directive, which is created by using the **@Component** decorator.

We can create our own Directives by using the **@Directive** decorater. Apart from @Component, the two other commonly used ones are.

- Structural directives
- Attribute directives

Structural directives adds logic to template.

```
<div *ngIf="users.length > 0">
    <p *ngFor="let user of users">
        {user.name}
    </p>
<div>
```

Here we use ***ngIf** and ***ngFor** to alter the logic based on the value of the users variable.

Noticed the scary * character prefixing ngFor? That is a shorthand, which expands to this:

```
<ng-template ngFor let-msg [ngForOf]="messages">
    <div>{{ msg }}</div>
</ng-template>
```

All structural directives, who change the HTML structure, uses the same * syntax.

Attribute directives alter the appearance or behavior of an existing element.It is commonly used to bind input fields to Model properties.

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

This binds hero.name to our input element.

COMMON DIRECTIVES

Lets take another look at the most common directives, and how we use them to render the view.

First ngFor, which as the name implies is a helper for rendering lists:

```
<div *ngFor="let msg of messages">{{ msg }}</div>
```

This will repeat the div for every item in the messages array.

Equally mandatory as ngFor is the ngIf directive, which lets you optionally display something given the truthiness of an expression.

```
<p *ngIf="flag">I only show when flag is true!</p>
```

This will repeat the div for every item in the messages array.

Then ngSwitch, which of course emulates a switch statement in the template. But! We're actually not going to go into detail, as ngSwitch is rarely useful.

Such logic often belongs in the model, a subject we'll come back to.

Now ngClass which can dynamically set a single style...

```
<div [class.someClass]="someCondition"></div>
```

...or many (keys are classes, values are conditions) :

```
<div [ngClass]="someObject"></div>
```

COMPONENT INTERACTION

Components can pass data between each other. A well structured application has many components, each solving exactly one problem.

Lets look at an example

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

# ChildComponent
export class ChildComponent {
@Input() user;
}

# ParentComponent
...
template: `
<child-component *ngFor="let user of users" [user]="user">
</child-component>
`
```

So we use @Input in ChildComponents to recieve input from a parent. What if we want to recieve data from a child instead?

We can use @Output in our ChildComponent, to create and trigger a custom event.

```
import { Component, EventEmitter, Input, Output } from '@angul

# ChildComponent
export class ChildComponent {
    @Output () changeEvent = new EventEmitter<boolean>();

    onChange() {
        this.changeEvent.emit(true);
    }
}
```

We can then listen to that event like any other in our Parent component.

```
# ParentComponent
template: `
<child-component (changeEvent)="logValue($event)"
</child-component>
`
...
export class ParentComponent {
    logValue(event) { console.log(event); }
}
```

Lets summarise the Component syntax that we have learned so far.

Component rendering

Scope	Purpose	Example
Interpolation	Renders expression	{{variableName}}
Event binding	Bind events to methods	(change)="method(variab

Directives		
Scope	Purpose	Example
*ngFor	Loops array	p *ngFor="let user of users"
*ngIf	Conditional rendering	p *ngIf="users.length > 0"

Component IO

Scope	Purpose	Example
@Input()	Defined in Component to take input from ParentComponents	@Input() user;
@Output()	Defined in Component to create a custom event	@Output customEvent = new EventEmitter();

EXERCISE 1

It's time to put this into practice

Jump to exercise