

AngularJS -> Angular

A complete rewrite



Press Space for next page →



Early Web

- Web designed for documents
- Server creates pages / browser displays
- Data input sent to and processed by the server
- Updated pages created on the server and resent

First Example - PHP

```
<!doctype html>
<html>
<head>
</head>
<body>
  <form method="post" action="hello.php">
    <label>Name:</label>
    <input type="text" id="yourName">
    <input type="submit" value="Say Hello" />
    <hr>
  <?php
    echo "<h1>Hello ". $HTTP_POST_VARS["yourName"]."!</h1>";
  ?>
  </form>
</body>
</html>
```

Web Evolution - AJAX

- Interactive client-side web
 - Collect input from user
 - Update display
 - Communicate with server
- Client-side processing enabled by
 - JavaScript
 - DOM manipulation
 - HTTP server messaging

First Example - jQuery

```
<html>
<head>
  <script
    src="http://code.jquery.com/jquery-1.9.1.min.js">
  </script>
  <script type="text/javascript">
    $(function() {
      $("#yourName").keyup(function () {
        $("#helloName").text("Hello " + this.value + "!");
      });
    });
  </script>
</head>
<body>
  <div>
    <label>Name:</label>
    <input type="text" id="yourName">
    <hr>
    <h1 id="helloName"></h1>
  </div>
</body>
</html>
```

Result

JavaScript

HTML

CSS



Edit in JSFiddle

Name:

jQuery

- Simplifies event binding and DOM manipulation
- Common API across multiple browsers
- Supports plug-in modules to extend functionality
- Requires writing JavaScript code to wire


Today's Web – Can we do better?

- Follow good programming practices
 - Separate: data / display / processing
 - Simplify connecting data to display
- Let us focus on the technologies of the web
 - HTML
 - CSS
 - JavaScript

First Example - Angular

```
main.ts
1 import 'zone.js';
2 import { Component } from '@angular/core';
3 import { bootstrapApplication } from '@angular/
  platform-browser';
4 import { FormsModule } from '@angular/forms';
5
6 @Component({
7   selector: 'app-root',
8   standalone: true,
9   imports: [FormsModule],
10  template: `
11    <h1>Hello from {{ name }}!</h1>
12    <input type="text" [(ngModel)]="name">
13  `,
14 })
15 export class App {
16   name = 'Angular';
17 }
```

Compiling application & starting dev server...

Edit on  StackBlitz

Hello from Angular!

Console

1

Editor

Preview

Both

Imperative vs. Declarative

```
<input type="text" id="yourName">
<h1 id="helloName"></h1>
<script type="text/javascript">
  $(function() {
    $("#yourName").keyup(function () {
      $("#helloName").text("Hello " + this.value + "!");
    });
  });
</script>
```

to Angular declarative relationships

```
<input type="text" [(ngModel)]="yourName">
<h1>Hello {{yourName}}!</h1>
```

Abstractions

- jQuery abstracts browser functionality
 - e.g. DOM traversal, event binding
- AngularJS/Angular abstracts relationships (and more)

AngularJS/Angular, and all web apps, are built on browser functionalities

The DOM abstraction

- HTML is a declarative document language
- Browser translates HTML into a Document Object Model (DOM)
- DOM is the browser's in-memory document representation
- JavaScript can manipulate the DOM

AngularJS/Angular "*compiles*" HTML

- Browsers send a document (i.e. DOM) ready event
- Angular/AngularJS can intercede and rewrite the DOM
- The rewrite is driven by markup in the DOM

Angular "*compiles*" HTML

Why complete rewrite?

- JavaScript is more matured now
- Typescript is more mature now
- Browsers are more matured now
- Performance improvements
- Saying bye to some of old friends (scope, modules, DDO, controllers, jqLite)
- And several other reasons...

First core concept: Web components (WC)

Web Components are not a single technology. Instead, they are series of browser standards defined by the W3C allowing developers to build components in a way the browser can natively understand. These standards include:

- **HTML Templates and Slots** – Reusable HTML markup with entry points for user-specific markup
- **Shadow DOM** – DOM encapsulation for markup and styles
- **Custom Elements** – Defining named custom HTML elements with specific behaviour
- **HTML import**

WC - Introduction

- Interest
 - Enrich the Web with new tags
 - The HTML standard has only about a hundred tags
 - Help with code reuse - avoid copy and paste
- Principles
 - Create new tags
 - Encapsulate the code in order to mask and isolate its complexity
 - Be able to import and declare tags in other projects/pages

WC - History

- Initiated by Google since 2010 with the Polymer project,
- relayed by Mozilla and other web actors
- Based on current standards at W3C
- Chrome 36 first compatible browser
- Polyfill" technology for older browsers
 - Library webcomponents.js replacing platform.js (since end 2014, with the transfer of the Polymer library to WebComponents.org)

WC - Custom element

- Instantiate a Custom Element
- Declaratively - the most elegant way

```
<script> document.registerElement('mon-composant'); </script>  
<mon-composant></mon-composant>
```

In the code via the constructor

```
var MyComponent = document.registerElement('mon-composant');  
var dom = new MyComponent();  
document.body.appendChild(dom);
```

In the code via document.createElement() - the most used solution in JS

```
document.registerElement('mon-composant');  
var dom = document.createElement('mon-composant');  
document.body.appendChild(dom);
```

WC - Custom Element Behavior

```
var proto = Object.create(HTMLElement.prototype);
proto.nom = 'Mon Composant';
proto.afficheNom = function() {
  console.log('Nom de la balise : ' + this.nom);
};
document.registerElement('mon-composant', {prototype: proto });
var dom = document.createElement('mon-composant');
```

WC - Custom Element Inheritance

- Inheriting an existing HTML tag
 - declare in *registerElement()* options with keyword *extends*
 - derive from the prototype of the inherited HTML element

```
document.registerElement('mon-champs-saisie', {  
  extends: 'input',  
  prototype: Object.create(HTMLInputElement.prototype)  
});
```

```
<input is="mon-champs-saisie"></input> <!-- Use -->
```

WC - Custom Element Lifecycle

- Prototype callbacks to help us define the life cycle
 - *createdCallback()* called after the creation of the element
 - *attachedCallback()* called when attaching to the DOM
 - *detachedCallback()* called during the secondment to the DOM
 - *attributeChangedCallback()* called during an attribute change

```
var proto = Object.create(HTMLElement.prototype);
proto.createdCallback = function() {
  var div = document.createElement('div');
  div.textContent = 'Le contenu de mon composant';
  this.appendChild(div);
};
document.registerElement('mon-composant', {prototype: proto});
```

WC - HTML Template

- Have a predefined reusable format
- Not having to recreate the same frame every time
- Concept existing in the web, but on the server side as well as
 - Apache Velocity in Java
 - Django in Python
 - Smarty in PHP
 - And many others...
- Few client-side solutions (executed in the browser)

WC - HTML Template

- A simple declaration using the tags
 - `<template>...</template>`
- Characteristics of this element
- Its content is not visible by the browser's rendering engine
- Scripts do not run, images are not loaded,...
- The content is not considered to be attached to the DOM
 - (`document.getElementById()` or `querySelector()` do not work)
- It can be placed anywhere in the HTML page

Activate a template

```
var t = document.querySelector('template');  
var clone = document.importNode(t.content, true);  
document.body.appendChild(clone);
```

WC - Shadow DOM

- Shadow DOM allows you to separate the content of the presentation while eliminating name conflicts
- Allows to hide all the internal kitchen of a component
- Allows you to encapsulate styles naturally

```
<style> p { color: Green; } </style>
<p>Du texte dans ma page html</p>
<div id="element"></div>
<script>
var foo = document.getElementById('element');
foo.createShadowRoot();
var p = document.createElement('p');
foo.shadowRoot.appendChild(p);
p.textContent = 'Du texte dans le shadow DOM';
</script>
```


WC - Shadow DOM

- The presentation is hidden in the Shadow DOM
- It must be possible to define and insert content
 - The tag `<content>` allows to identify insertion points
 - The text between the tags will be inserted instead of `<content>`

```
<style> p { color: Green; } </style>
<p>Du texte dans ma page html</p>
<div id="element">Olivier</div>
<script>
var foo = document.getElementById('element');
foo.createShadowRoot();
var p = document.createElement('p');
foo.shadowRoot.appendChild(p);
p.innerHTML = 'Mon nom est : <content/>';
</script>
```

HTML import - Introduction

- Custom Element, HTML Template and Shadow DOM allow to create Web Components, how to reuse them?
- You can currently load JS, CSS and HTML elements separately
- Imagine the complexity if you import Web Components that also use other Web Components Or use an iframe, or even JS code...

HTML import - Principle

- new definition of the link tag

```
<link rel='import' href='myComponent.html'>
```

- Allows to import the content of myComponent
 - href contains the path to the html file
- HTML import will load the HTML document, resolve the sub-resource loading and execute the JavaScript code.
 - The content is not automatically displayed at the import location
 - Code must be written to make this display
 - The rendering tags are not added to the DOM, but the style, script, link tags are well executed.

HTML import - rules

- The imported HTML file can load resources such as scripts, css, images... may not declare html, head, body, doctype tags.
- Several HTML imports referring to the same URL will only be imported and executed once
- Cross-domain restrictions apply see CORS (Cross Origin Resource Sharing) if necessary
- Ability to manage loading errors

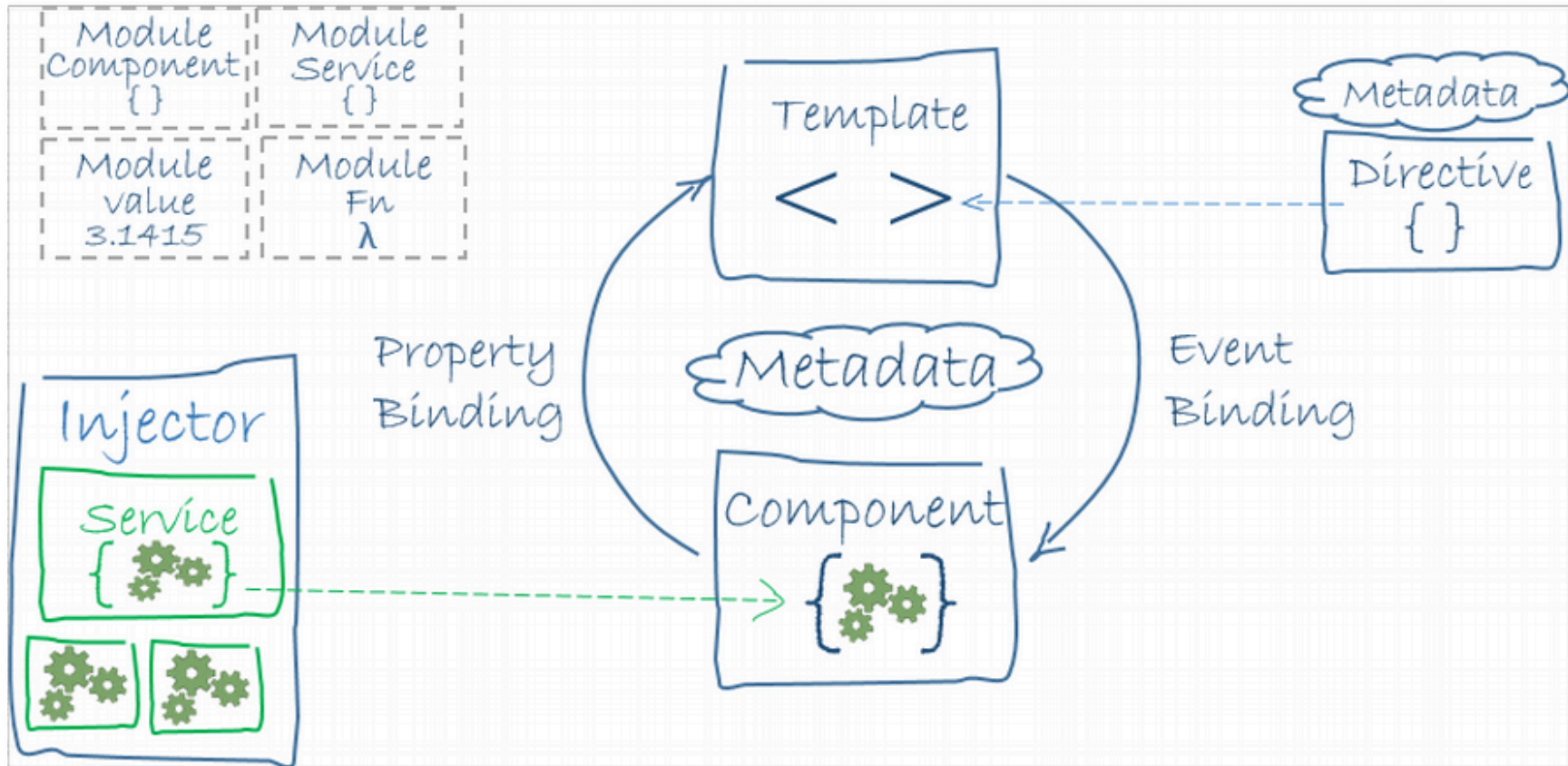
```
<link rel='import' href='myComponent.html '  
onload='handleLoad(event)'.  
onerror='handleError(event) '>
```

Angular

- V2. Release in September 2016
- Using TypeScript
- Many modern browsers support it (Even IE9)
- V9. Release Feb 6, 2020
- V18. Release 2024-05-22
- changelog

Big picture

Big picture



Angular concepts 1/2

- **Components**

- Application logic which controls parts of the user interface

- **Templates**

- Renders the component on the page

- **Data bindings**

- magic link between business model and template

- **Metadata** – Information about the angular application parts

- **Component interaction** – Information about the angular application parts

Angular concepts 2/2

- **Dependency Injection / Service**

- coding pattern in which a class asks for dependencies from external sources rather than creating them itself.

- **Routing**

- Navigation from one view to the next as users perform application tasks.

- **Forms**

- Handling user input

- **Pipe**

- Write display-value transformations that you can declare in your HTML.

- **Modules**

- Contains parts of the application which we export

- **Modules vs standalone mode**

- Contains parts of the application which we export

Angular concepts 1/2

- **Components**
 - Application logic which controls parts of the user interface
- **Templates**
 - Renders the component on the page
- **Data bindings**
 - magic link between business model and template
- **Metadata** – Information about the angular application parts
- **Signals**
 - Signal, effect and reactive programming
- **Component interaction**
 - Information about the angular application parts

Component

```
import {Component} from '@angular/core';
@Component({
  selector: 'my-app',
  template: `
    <h1>{{title}}</h1>
    <h2>My favorite hero is: {{myHero}}</h2>
  `
})
export class AppComponent {
  title = 'Tour of Heroes';
  myHero = 'Windstorm';
}
```

app.ts

```
<body>
  <my-app>Loading...</my-app>
</body>
```

index.html

Template: show the list of heroes

```
export class AppComponent {  
  title = 'Tour of Heroes';  
  heroes = ['Windstorm', 'Bombasto', 'Magenta', 'Tornado'];  
  myHero = this.heroes[0];  
}
```

Component

```
<h1>{{title}}</h1>  
<h2>My favorite hero is: {{myHero.name}}</h2>  
<p>Heroes:</p>  
<ul>  
  <li *ngFor="#hero of heroes">  
    {{ hero.name }}  
  </li>  
</ul>  
<p *ngIf="heroes.length > 3">There are many heroes!</p>
```

template

Example: create class for data and fill it

```
export class Hero {  
  constructor(  
    public id:number,  
    public name:string) { }  
}
```

hero.ts

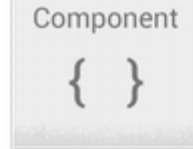
```
export class AppComponent {  
  title = 'Tour of Heroes';  
  heroes = [  
    new Hero(1, 'Windstorm'), new Hero(13, 'Bombasto'),  
    new Hero(15, 'Magenta'), new Hero(20, 'Tornado')  
  ];  
  myHero = this.heroes[0];  
}
```

app.component.ts:

Example: work with the events

```
@Component({
  selector: 'click-me',
  template: `
    <button (click)="onClickMe()">Click me!</button>
    {{clickMessage}}`
})
export class ClickMeComponent {
  clickMessage = '';
  onClickMe() {
    this.clickMessage = 'You are my hero!';
  }
}
```

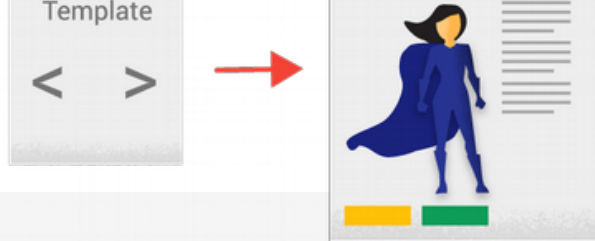
Component



```
export class HeroListComponent implements OnInit {  
  constructor(private _service: HeroService){ }  
  heroes: Hero[];  
  selectedHero: Hero;  
  ngOnInit(){  
    this.heroes = this._service.getHeroes();  
  }  
  selectHero(hero: Hero) { this.selectedHero = hero; }  
}
```

app/hero-list.component.ts

Template



```
<h2>Hero List</h2>

<p><i>Pick a hero from the list</i></p>
<div *ngFor="#hero of heroes" (click)="selectHero(hero)">
  {{hero.name}}
</div>

<hero-detail *ngIf="selectedHero" [hero]="selectedHero">
</hero-detail>
```

app/hero-list.component.html

foo

bar

1

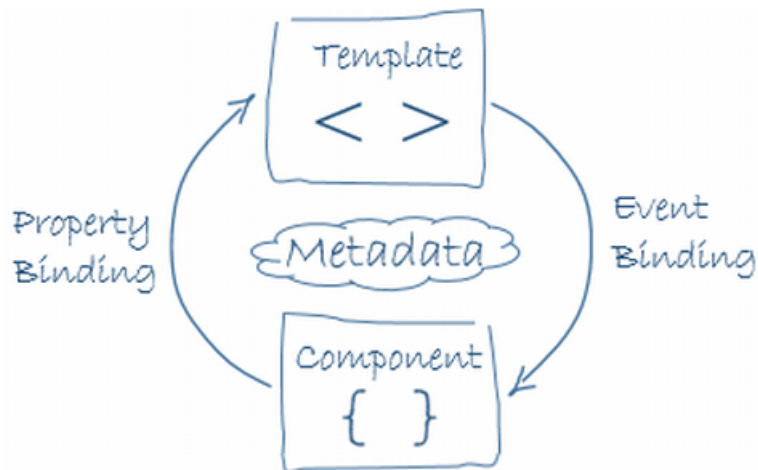
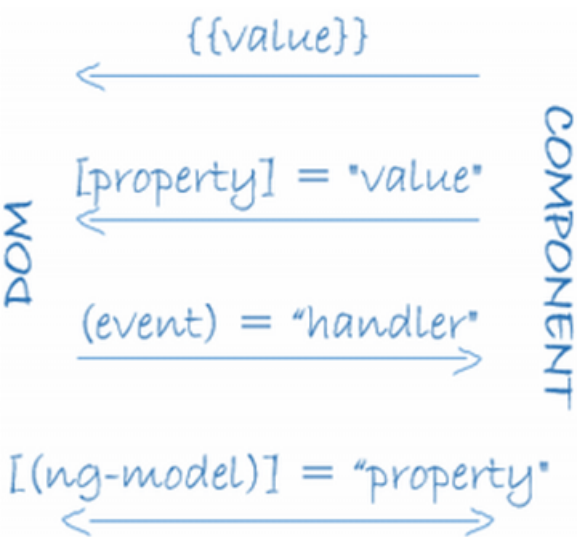
test2

cdc

Data binding

```
<div>{{hero.name}}</div>  
<hero-detail [hero]="selectedHero"></hero-detail>  
<div (click)="selectHero(hero)"></div>
```

app/hero-list.component.ts



Binding in templates

| Data Direction | Syntax | Binding Type |
|--|---|--|
| One way from data source to view target | <code>target = "expression"</code> <code>bind-target = "expr"</code> | Interpolation Property Attribute Class Style |
| One way from view target to data source | <code>(target) = "expression"</code> <code>on-target = "expr"</code> | Event |
| Two way | <code>(target) = "expr"</code> <code>bindon-target = "expr"</code> | Two-way |

Binding targets

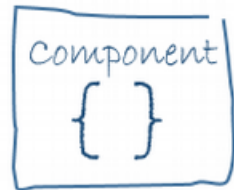
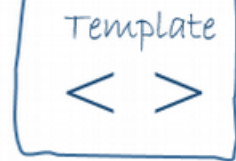
| Binding Type | Target | Examples |
|--------------|--------------------|---|
| Property | Element Property | <code></code> |
| | Component Property | <code><hero-detail [hero]="currentHero"></hero-detail></code> |
| | Directive property | <code><div [ngClass] = "{selected: isSelected}"></div></code> |
| Event | Element Event | <code><button (click) = "onSave()">Save</button></code> |
| | Component Event | <code><hero-detail (deleted)="onHeroDeleted()"/></code> |
| | Directive Event | <code><div mC(mC)="clicked=\$event">click me</div></code> |

Binding targets

| Binding Type | Target | Examples |
|--------------|------------------------------|--|
| Two-way | Directive Event Property | <code><input [(ngModel)]="heroName"></code> |
| Attribute | Attribute (the exception) | <code><button [attr.aria-label]="help"> help </button></code> |
| Class | class Property | <code><div [class.special]="isSpecial"> Special </div></code> |
| Style | style Property | <code><button [style.color] = "isSpecial ? 'red' : 'green' "></code> |

Metadata

```
@Component({  
  selector: 'hero-list',  
  templateUrl: 'app/hero-list.component.html',  
  directives: [HeroDetailComponent],  
  providers: [HeroService]  
})  
export class HeroesComponent { ... }
```



- **selector** : a css selector that tells Angular to create and insert an instance of this component where it finds a `<hero-list>` tag in parent HTML.

```
<hero-list></hero-list>
```

Angular inserts an instance of the HeroListComponent view between tags.

- **templateUrl**: the address of this component's template
- **directives**: an array of the Components or Directives this template requires
- **providers**: an array of dependency injection providers for services that the component requires

Signals

Angular Signals is a system that granularly tracks how and where your state is used throughout an application, allowing the framework to optimize rendering updates.

A signal is a wrapper around a value that notifies interested consumers when that value changes.

Signals can contain any value, from primitives to complex data structures.

Writable signals

Writable signals provide an API for updating their values directly. You create writable signals by calling the `signal` function with the signal's initial value:

```
const count = signal(0);  
// Signals are getter functions - calling them reads their value.  
console.log('The count is: ' + count());
```

To change the value of a writable signal, either `.set()` it directly:

```
count.set(3);
```

or use the `.update()` operation to compute a new value from the previous one:

```
// Increment the count by 1.  
count.update(value => value + 1);
```

Computed signals

Computed signal are read-only signals that derive their value from other signals. You define computed signals using the `computed` function and specifying a derivation:

```
const count: WritableSignal<number> = signal(0);  
const doubleCount: Signal<number> = computed(() => count() * 2);
```

The `doubleCount` signal depends on the `count` signal. Whenever `count` updates, Angular knows that `doubleCount` needs to update as well.

- Computed signals are both lazily evaluated and memoized
- Computed signals are not writable signals

Signals and template

When you use a signal within a template interpolation, you need to invoke it to render its value.

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

```
@Component({
```

```
  // ...
```

```
})
```

```
class MyComponent {
```

```
  mySignal: Signal = signal(0)
```

```
}
```

```
<div>{{ mySignal() }}/div>
```

Effects

Signals are useful because they notify interested consumers when they change. An effect is an operation that runs whenever one or more signal values change. You can create an effect with the effect function:

```
effect(() => {  
  console.log(`The current count is: ${count()}`);  
});
```

Signals summary

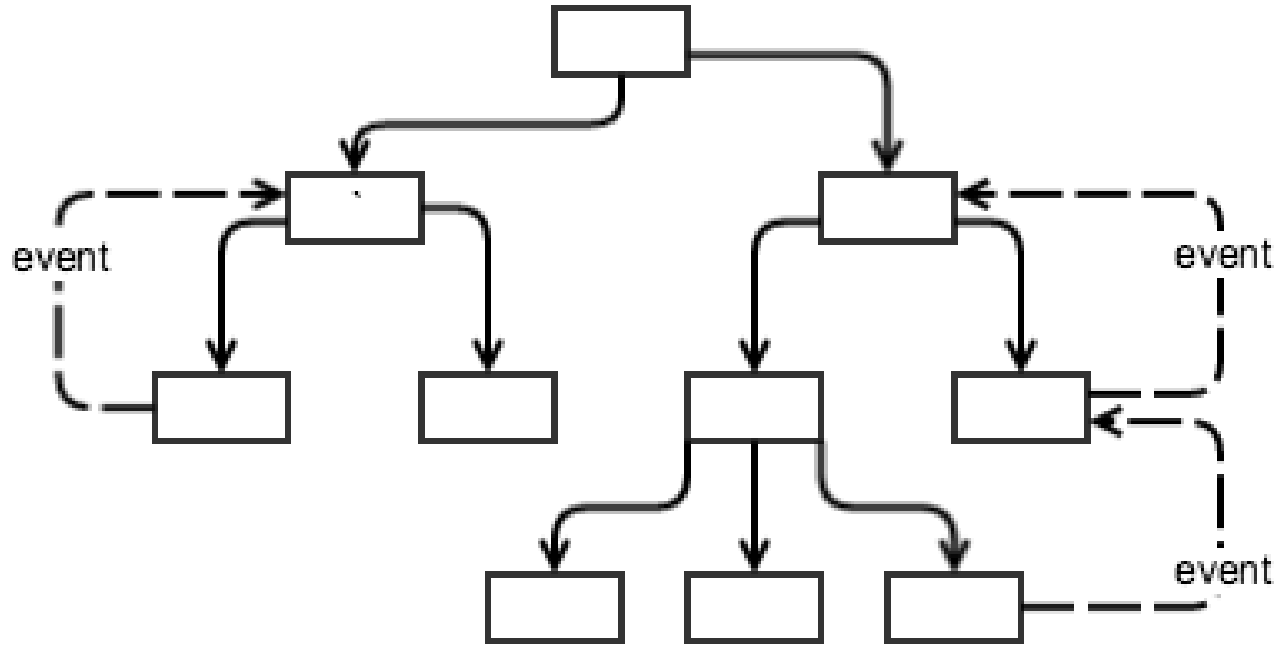
Signals are:

- Not a Replacement for RxJS or Observables for async operations such as `http.get`
- Reactive in Nature
- Always have a value
- Synchronous

Can be used in Components, Services, Templates, or Directives

Component interaction

Component interaction



Component interaction

“Inputs”, as you might guess from the hierarchy discussion above, specifies which properties you can set on a component whereas “outputs” identifies the events a component can fire to send information up the hierarchy to its parent.

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

@Component({
  selector: 'child-comp',
  template: `
    <div class="child">
      <h2>Child component</h2>
      <button (click)="updateCount()">Add To Parent</button>
    </div>`})

export class ChildComponent {
  @Input("parentCount")
  count: number;

  @Output()
  change: EventEmitter<number> = new EventEmitter<number>();

  updateCount() {
    this.count++;
    this.change.emit(this.count);
  }
}
```

Component interaction

```
@Component({
  selector: 'my-app',
  template: `
    <div>
      <button (click)="reset()">Reset count</button>
      <div>
        <h2>Parent component</h2>
        <div>
          count from child: {{count}}
        </div>
        <child-comp
          [parentCount]="count"
          (change)="updateFromChild($event)"></child-comp>
      </div>
    </div>`
})
export class App {
  count: number = 0;
  updateFromChild($event){
    console.log($event);
  }
  reset(){
    this.count = 0;
  }
}
```

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

@Component({
  selector: 'child-comp',
  template: `
    <div class="child">
      <h2>Child component</h2>
      <button (click)="updateCount()">Add To Parent</button>
    </div>`
})
export class ChildComponent {
  @Input("parentCount")
  count: number;

  @Output()
  change: EventEmitter<number> = new EventEmitter<number>();

  updateCount() {
    this.count++;
    this.change.emit(this.count);
  }
}
```

Angular concepts 2/2

- **Dependency Injection / Service**

- coding pattern in which a class asks for dependencies from external sources rather than creating them itself.

- **Routing**

- Navigation from one view to the next as users perform application tasks.

- **Forms**

- Handling user input

- **Pipe**

- Write display-value transformations that you can declare in your HTML.

- **Modules**

- Contains parts of the application which we export

- **Modules vs standalone mode**

- Contains parts of the application which we export

Dependency Injection

```
constructor(private _service: HeroService) { ... }
```

But how Injector get to know about service?

2 ways:

```
bootstrap(AppComponent, [HeroService, Logger]);
```

app/boot.ts

```
@Component({  
  providers: [HeroService]  
})  
export class HeroesComponent { ... }
```

app/hero-list.component.ts

(we get a new instance of the service with each new instance of that component)

Component *service*
{Constructor(service)}

Service

```
export class HeroService {  
  constructor(private _backend:BackendService,  
               private _logger:Logger) {}  
  private _heroes:Hero[] = [];  
  
  getHeroes() {  
    this._backend.getAll(Hero).then((heroes:Hero[]) => {  
      this._logger.log(`Fetched ${heroes.length} heroes.`);  
      this._heroes.push(...heroes); // fill cache  
    });  
    return this._heroes;  
  }  
}
```

app/hero.service.ts

Example: define mock HeroService

```
import {HEROES} from './mock-heroes';
import {Injectable} from '@angular/core';

@Injectable()
export class HeroService {
  getHeroes() {
    return HEROES;
  }
}
```

hero.service.ts:

```
import {Hero} from './hero';
export var HEROES: Hero[] = [
  {"id": 11, "name": "Mr. Nice"},
  {"id": 12, "name": "Narco"}
]
```

mock-heroes.ts

Example: define HeroService with promises

```
import {Injectable} from '@angular/core';
import {Hero} from './hero';
import {HEROES} from './mock-heroes';

@Injectable()
export class HeroService {
  getHeroes() {
    return Promise.resolve(HEROES);
  }
  // See the "Take it slow" appendix
  getHeroesSlowly() {
    return new Promise<Hero[]>(resolve =>
      setTimeout(()=>resolve(HEROES), 2000) // 2 seconds
    );
  }
}
```

Example: use HeroService in component

```
import {Component, OnInit} from '@angular/core';
import {HeroService} from './hero.service';
@Component({
  selector: 'my-app',
  template: `...`,
  providers: [HeroService]
})
export class AppComponent implements OnInit {
  heroes:Hero[];
  constructor(private _heroService:HeroService) { }
  getHeroes() {
    this._heroService.getHeroes()
      .then(heroes => this.heroes = heroes);
  }
  ngOnInit() {    this.getHeroes();    }
}
```

Routing

```
@Component({...})
@RouteConfig([
  { path: '/crisis-center',
    name: 'CrisisCenter',
    component: CrisisListComponent },
  { path: '/heroes',
    name: 'Heroes',
    useAsDefault: true,
    component: HeroListComponent },
  { path: '/hero/:id',
    name: 'HeroDetail',
    component: HeroDetailComponent }
])
export class AppComponent {}
```

```
<!-- Routed views go here -->
<router-outlet></router-outlet>
```

Routing: defining in main app component

```
import { RouteConfig, ROUTER_DIRECTIVES, ROUTER_PROVIDERS }
  from '@angular/router';

@Component({
  selector: 'my-app',
  template: `
    <a [routerLink]="['Heroes']">Heroes</a>
    <a [routerLink]="['CrisisCenter']">Crisis Center</a>
    <router-outlet></router-outlet> `,
  directives: [ROUTER_DIRECTIVES],
  providers: [ ROUTER_PROVIDERS, HeroService ]
})
@RouteConfig([
  { path: '/heroes', name: 'Heroes', component: HeroesComponent }
])
export class AppComponent {}
```

Routing: heroes component

```
@Component({
  selector: 'my-heroes',
  templateUrl: 'app/heroes.component.html',
  styleUrls: ['app/heroes.component.css'],
  directives: [HeroDetailComponent]
})
export class HeroesComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;
  constructor( private _router: Router,
    private _heroService: HeroService) {}
  getHeroes() {
    this._heroService.getHeroes().then(heroes =>
      this.heroes = heroes);
  }
  ngOnInit() { this.getHeroes(); }
  onSelect(hero: Hero) { this.selectedHero = hero; }
  gotoDetail() {
    this._router.navigate(['HeroDetail', { id:
      this.selectedHero.id }]);
  }
}
```


Routing: heroes template

```
<h2>My Heroes</h2>
<ul class="heroes">
  <li *ngFor="#hero of heroes"
      [class.selected]="hero === selectedHero"
      (click)="onSelect(hero)">
    <span class="badge">{{hero.id}}</span> {{hero.name}}
  </li>
</ul>

<div *ngIf="selectedHero">
  <h2>
    {{selectedHero.name | uppercase}} is my hero
  </h2>
  <button (click)="gotoDetail()">View Details</button>
</div>
```

MR. NICE is my hero

View Details

Routing: hero details component

```
@Component({
  selector: 'my-hero-detail',
  templateUrl: 'app/hero-detail.component.html',
})
export class HeroDetailComponent implements OnInit {
  hero: Hero;
  constructor(
    private _heroService: HeroService,
    private _routeParams: RouteParams) {
  }
  ngOnInit() {
    let id = +this._routeParams.get('id');
    this._heroService.getHero(id)
      .then(hero => this.hero = hero);
  }
}
```

app/hero-detail.component.ts

Routing: hero details template

```
<div *ngIf="hero">
  <h2>{{hero.name}} details!</h2>
  <div>
    <label>id: </label>{{hero.id}}</div>
    <div>
      <label>name: </label>
      <input [(ngModel)]="hero.name" placeholder="name" />
    </div>
    <button (click)="goBack()">Back</button>
  </div>
```

app/hero-detail.component.html

Forms

```
<div class="form-group">
  <label for="power">Hero Power</label>
  <select class="form-control" required>
    <option *ngFor="#p of powers" [value]="p">{{p}}</option>
  </select>
</div>
```

NgControl

```
<input type="text" class="form-control "
  required
  [(ngModel)]="model.name"
  ngControl="name" >
```

Form validation

| State | Class if true | Class if false |
|-----------------------------|---------------|----------------|
| Control has been visited | ng-touched | ng-untouched |
| Control's value has changed | ng-dirty | ng-pristine |
| Control's value is valid | ng-valid | ng-invalid |

```
.ng-valid[required] {  
  border-left: 5px solid #42A948; /* green */  
}  
.ng-invalid {  
  border-left: 5px solid #a94442; /* red */  
}
```

Show validation messages

```
<input type="text" class="form-control" required  
  [(ngModel)]="model.name"  
  ngControl="name" #name="ngForm" >  
<div [hidden]="name.valid" class="alert alert-danger">  
  Name is required  
</div>
```

Using pipes

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

@Component({
  selector: 'hero-birthday',
  template: `<p>The hero's birthday is
             {{ birthday | date }}</p>`
})
export class HeroBirthday {
  birthday = new Date(1988,3,15); // April 15, 1988
}
```

Define custom pipe

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

/* Raise the value exponentially
 * Takes an exponent argument that defaults to 1.
 * Usage: value | exponentialStrength:exponent
 * Example:
 *   {{ 2 | exponentialStrength:10}}
 *   formats to: 1024
 */
@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe {
  transform(value:number, args:string[]) : any {
    return Math.pow(value, parseInt(args[0] || '1', 10));
  }
}
```


Use custom pipe

```
import {Component} from '@angular/core';
import {ExponentialStrengthPipe} from './exponential-strength.pipe';

@Component({
  selector: 'power-boost',
  template: `
    <h2>Power Booster</h2>
    <p>
      Super power boost: {{2 | exponentialStrength: 10}}
    </p>
  `,
  pipes: [ExponentialStrengthPipe]
})
export class PowerBooster { }
```

Use Async pipe

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

// Initial view: "Message: "
// After 500ms: Message: You are my Hero!"

@Component({
  selector: 'hero-message',
  template: 'Message: {{delayedMessage | async}}',
})
export class HeroAsyncMessageComponent {
  delayedMessage: Promise<string> =
    new Promise((resolve, reject) => {
      setTimeout(() => resolve('You are my Hero!'), 500);
    });
}
```

Modules VS Standalone components

Modules

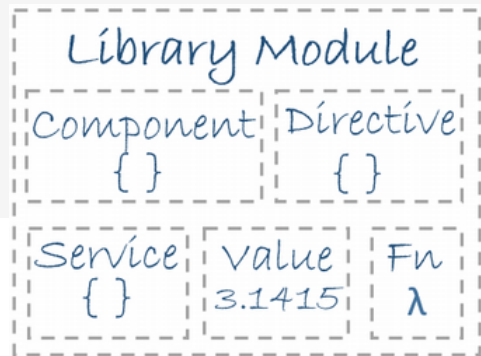
Angular defines modules as containers for a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities. They can contain components, service providers, and other code files whose scope is defined by the containing module. They can import functionality that is exported from other modules, and export selected functionality for use by other modules.

equivalent to decide which classes can be imported from a jar (Public Private for a module)

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

- Angular apps are composed of modules.
- Modules export things — classes, function, values — that other modules import.
- We prefer to write our application as a collection of modules, each module exporting one thing.

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



Standalone component

- Introduced in Angular 15
- eliminate the requirement for modules.

With modules

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.scss'],  
})  
export class AppComponent {}
```

With standalone component

```
@Component({  
  selector: 'app-root',  
  standalone: true,  
  imports: [RouterOutlet],  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.scss'],  
})  
export class AppComponent {}
```

- Reduced bundle size and improved build times
- Enhanced lazy loading and routing capabilities

```
export const ROUTES: Route[] = [  
  {path: 'foo', loadComponent: () =>  
    import('./foo/bar.component').then(mod => mod.BarComponent)},  
];
```