ANGULAR

1. [Why Angular](#_Why_Angular)
2. [Angular Bootstrapping](#_Angular_Bootstrapping)
3. [Component Life cycle](#_Component_Life_cycle)
4. Data Flow b/w components
5. [Parent to Child](#_Data_Flow_b/w)
6. [Child to Parent](#_Data_Flow_b/w_1)
7. [Directive](#_Directive)
8. [Http Interceptor](#_Http_Interceptor)
9. [Promise vs. Observable](#_Promise_vs._Observable)
10. [HOT vs. Cold Observables](#_HOT_vs._Cold)
11. [Observable Subjects](#_Observable_Subjects)
12. JIT vs. AOT

# Why Angular

* Mvc Architecture support
* Two way binding support
* Default Ivy renderer - optimized bundle size, faster component loading
* Di support
* Modular
* Component based development
* JIT/AOT compilation support
* Support SPA
* Declarative user interface
* Google support
* Easy integration
* Angular integration is pre-built into several frameworks, for example, Ionic, Telerik’s Kendo UI, Wijmo, etc. Hence, integration of third-party features is easy with Angular. So, if you want to add some high-quality user interface components, you can easily do so by using any of the above-mentioned frameworks!
* Cross-Platform
* Angular can be used to make any of the following types of applications:
  + Web applications: Angular can be used for web development. Also, from Angular 5 onwards, progressive web applications can also be developed. Such applications have high performance and can work offline as well.
  + Native mobile applications: Native mobile applications can be built using Angular.
* Desktop applications: Angular can be used to create desktop-installed applications for Mac, Windows, and Linux.

# Angular Bootstrapping

The process of loading the **index.html** page, app-level module, and app-level component is called bootstrapping, or loading the app. In this guide, you will learn about the internals of the bootstrapping process. Angular takes the following steps to bootstrap the application:

1. Load index.html
2. Load Angular, Other Libraries, and App Code
3. Execute ***main.ts*** File
4. Load App-Level Module
5. Load App-Level Component
6. Process Template

# Component Life cycle

* **Constructor()**
* **ngOnChanges( )** This hook/method is called before ngOnInit and whenever one or more input properties of the component changes. This method/hook receives a SimpleChanges object which contains the previous and current values of the property.
* **ngOnInit( )** This hook gets called once, after the ngOnChanges hook.It initializes the component and sets the input properties of the component.
* **ngDoCheck( )** It gets called after ngOnChanges and ngOnInit and is used to detect and act on changes that cannot be detected by Angular. We can implement our change detection algorithm in this hook.
* **ngAfterContentInit( )** It gets called after the first ngDoCheck hook. This hook responds after the content gets projected inside the component.
* **ngAfterContentChecked()** It gets called after ngAfterContentInit and every subsequent ngDoCheck. It responds after the projected content is checked.
* **ngAfterViewInit()** It responds after a component's view, or a child component's view is initialized.
* **ngAfterViewChecked()** It gets called after ngAfterViewInit, and it responds after the component's view, or the child component's view is checked.
* **ngOnDestroy()** It gets called just before Angular destroys the component. This hook can be used to clean up the code and detach event handlers.

# Data Flow b/w Components: Parent to Child

## Using @Input decorator:

Consider the following parent component:

@Component({

selector: 'app-parent',

template: `

<app-child [data]=data></app-child>

` ,

styleUrls: ['./parent.component.css']

})

export class ParentComponent{

data:string = "Message from parent";

constructor() { }

}

In the above parent component, we are passing “data” property to the following child component:

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

@Component({

selector: 'app-child',

template:`

<p>{{data}}</p>

`,

styleUrls: ['./child.component.css']

})

export class ChildComponent {

@Input() data:string

constructor() { }

}

In the child component, we are using @Input decorator to capture data coming from a parent component and using it inside the child component’s template.

# Data Flow b/w Components: Child to Parent

## Using @ViewChild decorator

**Child component:**

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

@Component({

selector: 'app-child',

template:`

<p>{{data}}</p>

`,

styleUrls: ['./child.component.css']

})

export class ChildComponent {

data:string = "Message from child to parent";

constructor() { }

}

**Parent Component:**

import { Component,ViewChild, AfterViewInit} from '@angular/core';

import { ChildComponent } from './../child/child.component';

@Component ({

selector: 'app-parent',

template: `

<p>{{dataFromChild}}</p>

` ,

styleUrls: ['./parent.component.css']

})

export class ParentComponent implements AfterViewInit {

dataFromChild: string;

@ViewChild(ChildComponent,{static:false}) child;

**ngAfterViewInit(){**

this.dataFromChild = this.child.data;

}

constructor () { }

}

In the above example, a property named “data” is passed from the child component to the parent component.

**@ViewChild** decorator is used to reference the child component as “child” property. Using the **ngAfterViewInit** hook; we assign the child’s data property to the *messageFromChild* property and use it in the parent component’s template.

## Using @Output and EventEmitter

In this method, we bind a DOM element inside the child component, to an event (click event for example) and using this event we emit data that will captured by the parent component:

**Child Component:**

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

@Component({

selector: 'app-child',

template:`

<button (click)="emitData()">Click to emit data</button>

`,

styleUrls: ['./child.component.css']

})

export class ChildComponent {

data:string = "Message from child to parent";

@Output() dataEvent = new EventEmitter<string>();

constructor() { }

emitData(){

this.dataEvent.emit(this.data);

}

}

As you can see in the child component, we have used ***@Output***property to bind an EventEmitter. This event emitter emits data when the button in the template is clicked.

**In the parent component’s template we can capture the emitted data like this:**

**<app-child (dataEvent) ="receiveData ($event)"></app-child>**

Then inside the receiveData function we can handle the emitted data:

receiveData($event){

this.dataFromChild = $event;

}

# Directive

A directive is a class in Angular that is declared with a @Directive decorator.

Every directive has its own behavior and can be imported into various components of an application.

## Types of directives

**Component directives:**

These form the main class in directives. Instead of @Directive decorator we use *@Component* decorator to declare these directives. These directives have a view, a stylesheet and a selector property.

**Structural directives**

These directives are generally used to manipulate DOM elements.

Every structural directive has a ‘\*’ sign before them.

We can apply these directives to any DOM element.

Let’s see some built-in structural directives in action:

<div \*ngIf="isReady" class="display\_name">

{{name}}

</div>

<div class="details" \*ngFor="let x of details" >

<p>{{x.name}}</p>

<p> {{x.address}}</p>

<p>{{x.age}}</p>

</div>

In the above example, we can \*ngIf and \*ngFor directives being used.

\*ngIf is used to check a boolean value and if it’s truthy,the div element will be displayed.

\*ngFor is used to iterate over a list and display each item of the list.

**Attribute Directives**

These directives are used to change the look and behavior of a DOM element. Let’s understand attribute directives by creating one:

How to create a custom directive?

We’re going to create an attribute directive:

In the command terminal, navigate to the directory of the angular app and type the following command to generate a directive:

ng g directive blueBackground

The following directive will be generated. Manipulate the directive to look like this:

import { Directive, ElementRef } from '@angular/core';

@Directive({

selector: '[appBlueBackground]'

})

export class BlueBackgroundDirective {

constructor(el:ElementRef) {

el.nativeElement.style.backgroundColor = "blue";

}

}

Now we can apply the above directive to any DOM element:

**<p appBlueBackground>Hello World!</p>**

# Http Interceptor

*"Interceptors in Angular, as the name suggests, is a simple way provided by the framework to intercept and modify the application’s http requests globally before they are sent to the server. That really comes in handy, allowing us to configure authentication tokens, add logs of the requests, add custom headers that out application may need and much more."*

**Implementing an Interceptor**

In order to implement an Interceptor, you need to create a class that implements the intercept method of the *HttpInterceptor* interface.

So let’s suppose you want to log in the console every single http request made by the application. Below, I’ve created a simple interceptor that would achieve the following.

@Injectable()

export class RequestLogInterceptor implements HttpInterceptor {

intercept(

request: HttpRequest<any>, next: HttpHandler

) : Observable<HttpEvent<any>> {

console.log(request.url);

return next.handle(request);

}

}

## Hooking up the Interceptor

To hook up the interceptor open up **app.module.ts**and assign the interceptor to the providers section.

Make sure to import the **HTTP\_INTERCEPTORS** at the top:

import {HttpClientModule, HTTP\_INTERCEPTORS} from '@angular/common/http';

and then add the interceptor(s) to the providers section:

providers: [

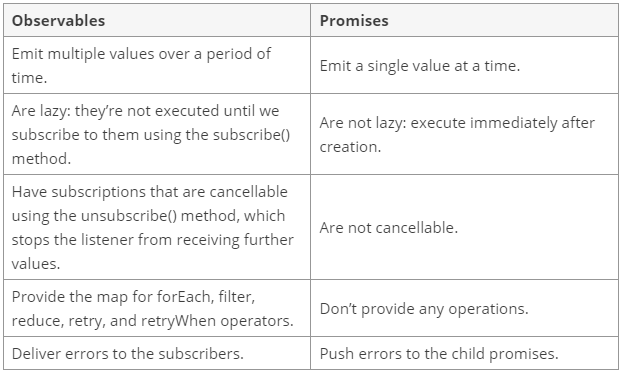
[

{ provide: HTTP\_INTERCEPTORS, useClass: RequestLogInterceptor, multi: true }

],

],

# Promise vs. Observable



# HOT vs. Cold Observables

***“When the data is produced by the Observable itself, we call it a cold Observable. When the data is produced outside the Observable, we call it a hot Observable.”***

## Cold Observable:

|  |  |
| --- | --- |
|  | import \* as Rx from ‘rxjs’  const random = Math.random() |
|  |  |
|  | const observable = Rx.Observable.create((observer) => { |
|  | observer.next(random); |
|  | }); |
|  |  |
|  | // subscription 1 |
|  | observable.subscribe((data) => { |
|  | console.log(data); // 0.11208711666917925 (random number) |
|  | }); |
|  |  |
|  | // subscription 2 |
|  | observable.subscribe((data) => { |
|  | console.log(data); // 0.11208711666917925 (random number) |
|  | }); |

## Hot Observables

import \* as Rx from ‘rxjs’

|  |  |
| --- | --- |
|  | const observable = Rx.Observable.fromEvent(document, 'click'); |
|  |  |
|  | // subscription 1 |
|  | observable.subscribe((event) => { |
|  | console.log(event.clientX); // x position of click |
|  | }); |
|  |  |
|  | // subscription 2 |
|  | observable.subscribe((event) => { |
|  | console.log(event.clientY); // y position of click |
|  | }); |

# Observable Subjects