**Data types Data can be** stored and manipulated **within a program**

**Primitive (Primary) 🡪 String, Number, Boolean**

**Composite*(*Reference*)* 🡪 Object, Array, Function**

**Specialdata types 🡪 Undefined, Null**

**Stringify – Parsing**

**JSON.stringify()**  takes a JavaScript object and then transforms it into a JSON string. Object🡪 String

**JSON.parse()**  takes a JSON string and then transforms it into a JavaScript object.(**Remove Colons**). String 🡪 Object

Method 1 :

// Store JSON data in a JS variable

var json = '{"name": "Peter", "age": 22, "country": "United States"}';

// Converting JSON-encoded string to JS object

var obj = JSON.parse(json);

// Accessing individual value from JS object

alert(obj.name); // Outputs: Peter

alert(obj.age); // Outputs: 22

alert(obj.country); // Outputs: United States

Method 2 :

/\* Storing multi-line JSON string in a JS variable using the new ES6 template literals \*/

var json = `{ "book": { "name": "Harry Potter and the Goblet of Fire", "author": "J. K. Rowling", "year": 2000, "characters": ["Harry Potter", "Hermione Granger", "Ron Weasley"], "genre": "Fantasy Fiction", "price": { "paperback": "$10.40", "hardcover": "$20.32", "kindle": "$4.11" } } }`;

// Converting JSON object to JS object

var obj = JSON.parse(json);

// Define recursive function to print nested values

function printValues(obj) {

for(var k in obj) {

if(obj[k] instanceof Object) {

printValues(obj[k]);

} else {

document.write(obj[k] + "<br>");

};

} };

// Printing all the values from the resulting object

printValues(obj);

document.write("<hr>");

// Printing a single value

document.write(obj["book"]["author"] + "<br>"); // Prints: J. K. Rowling document.write(obj["book"]["characters"][0] + "<br>"); // Prints: Harry Potter document.write(obj["book"]["price"]["hardcover"]); // Prints: $20.32

**try...catch**

JavaScript provides the try-catch statement to trap the runtime errors, and handle them gracefully.

try {  
    // Code that may cause an error  
} catch(error) {  
    // Action to be performed when an error occurs  
}

try { var greet = "Hi, there!"; document.write(greet);

// Trying to access a non-existent variable

document.write(welcome);

// If error occurred following line won't execute

alert("All statements are executed successfully.");

} catch(error) {

// Handle the error alert("Caught error: " + error.message);

} // Continue execution

document.write("<p>Hello World!</p>");

**Throwing Errors**

possible to throw an error manually by using the throw statement.

throw 123;

throw "Missing values!";

throw true;

throw { name: "InvalidParameter", message: "Parameter is not a number!" };

throw new Error("Something went wrong!");

**Boolean ( true/false)**

var a = true;

console.log(a); //true

var a = 17, b = 18, c = 20;

if(a<b){

console.log('not eligible'); //false

}else if(a<c){

console.log('not eligible'); //false

}else{

console.log('not eligible');

}

**Null (empty)**

(Ex: water 🡪 empty 🡪 Milk )

var a = "Jhon";

var b = "Doe";

console.log(a+' '+b);

console.log('before null'+' '+a);

a=null; //empty

a='new Jhon';

console.log('after null'+' ' +a);

emptyArray= [ ] //empty array

emptyObject={} //empty Object

**Undefined** ( value not assigned to variable)

var c;

console.log(c);

console.log(d); // d is *not defined* in object/document

**Object (** Store collections of data types. **)**

(Ex: Reference { employes : role } )

var a = {

x:10,

y:5,

z: function(){

return this.x+ '' +this.y;

},

b:function(){

return this.x+ '' +this.y;

}

}

console.log(a.y);

console.log(a.z());

console.log(a.b() + a.z());

delete a.y; // Deleting property

a.a(); // Outputs: Peter

a["z"](); // Outputs: Peter

// Iterating over object properties

for(var i in a) {

document.write(a[i] + "<br>"); // Prints: x, y,z and b

}

// Setting a new property

a.country = "United States";

document.write(a.country); // Prints: United States

// Assign person variable to a new variable

var user = a;

user.name = "Harry";

document.write(user.name); // Prints: Harry

**Functions (**group of statements**)(** Print return value **)**

* **Functions reduces the repetition of code within a program**
* **Functions makes the code much easier to maintain**
* **Functions makes it easier to eliminate the errors**

**Returning Values from a Function**

// Defining function

function getSum(num1, num2) {

var total = num1 + num2;

return total;

}

// Displaying returned value

alert (getSum(6, 20)); // 0utputs: 26

// Function Declaration

function getSum(num1, num2) {

var total = num1 + num2;

return total;

}

**Working with Function Expressions**

// Function Expression

var getSum = function(num1, num2) {

var total = num1 + num2;

return total;

};

var sum = getSum(7, 25);

alert(sum); // 0utputs: 32

**Defining and Calling a Function**

// Defining function

function sayHello() {

alert("Hello, welcome to this website!");

}

// Calling function

sayHello(); // 0utputs: Hello, welcome to this website!

// Function Expression

var greeting = function(){

return "Hello World!";

}

alert(type of greeting) // function

**Adding Parameters to Functions**

// Defining function

function displaySum(num1, num2) {

var total = num1 + num2;

alert(total);

}

// Calling function

displaySum(6, 20); // 0utputs: 26

**Default Values for Function Parameters**

function sayHello(name = 'Guest') {

alert('Hello, ' + name);

}

sayHello(); // 0utputs: Hello, Guest

**Arrays (**store more than one value/group of values**)(** Print return value**)**

1. **Single array**
2. **Multiple array**
3. **Associated array**

var myArray = [*element0*, *element1*, ..., *elementN*];

var fruits = ["Apple", "Banana", "Mango", "Orange", "Papaya"]; // Iterates over array elements

for(var i = 0; i < fruits.length; i++) {

document.write(fruits[i] + "<br>"); // Print array element

}

pop() 🡪 To remove the last element from an array

shift() 🡪 To remove the first element from an array

splice() 🡪 To add or remove elements from any index

( arr.splice(startIndex, deleteCount, elem1, ..., elemN) )

join() 🡪 To create a string by joining the elements of an array

slice( ) 🡪 To extract out a portion of an array (arr.slice(startIndex, endIndex))

concat() 🡪 To merge or combine two or more arrays

indexOf() 🡪 To the method returns the first one found and To search an array for a specific value

lastIndexOf() 🡪 To the lastIndexOf() returns the last one found

includes() 🡪 This method to find out whether an array includes a certain element or not

find() 🡪 findIndex() 🡪 This method returns the value of the first element

filter()🡪 This method to find out all the matched elements

sort() 🡪 for sorting array elements in alphabetical order

reverse() 🡪 To reverse the order of the elements of an array.

apply() 🡪This method in combination with the Math.max() and Math.min() to find the maximum and minimum value inside an array

## JavaScript Arithmetic Operators

## + - \* / %

## JavaScript Assignment Operators

## = += -= \*= /= %=

## JavaScript String Operators

## + +=

## JavaScript Incrementing and Decrementing Operators

## ++x x++ --x x—

## JavaScript Logical Operators

## && || !

## JavaScript Comparision Operators

## == === != !== < > >= <=

## Different type of Loops in Javascript

## while — loops through a block of code as long as the condition specified evaluates to true.

1. **do…while** — loops through a block of code once; then the condition is evaluated. If the condition is true, the statement is repeated as long as the specified condition is true.
2. **for** — loops through a block of code until the counter reaches a specified number.
3. **for…in** — loops through the properties of an object.
4. **for…of** — loops over iterable objects such as arrays, strings, etc.

## Closures

## This is accomplished by creating a function inside another function. closures are the primary mechanism used to enable data privacy.

## function makeCounter() {

## var counter = 0; // Nested function

## function make() {

## counter += 1;

## return counter;

## }

## return make;

## }

## /\* Execute the makeCounter() function and store the returned value in the myCounter variable \*/

## var myCounter = makeCounter();

## console.log(myCounter()); // Prints: 1

## console.log(myCounter()); // Prints: 2

## Creating the Getter and Setter Functions

## we will create a variable secret and protect it from being directly manipulated from outside code using closure.

## var getValue, setValue;

## // Self-executing function

## (function() {

## var secret = 0;

## // Getter function

## getValue = function() { return secret; };

## // Setter function

## setValue = function(x) {

## if(typeof x === "number") {

## secret = x; } };

## }());

## // Calling the functions

## getValue(); // Returns: 0

## setValue(10);

## getValue(); // Returns: 10

## setValue(null);

## getValue(); // Returns: 10

## Function Hoisting

## function before it is defined, but the code still works.

## // Calling function before declaration

## sayHello(); // Outputs: Hello, I'm hoisted!

## function sayHello() {

## alert("Hello, I'm hoisted!");

## }

## alert(str); // Outputs: undefined

## var str; // Declare and initialize

## str = "Hello World!";

**Local storage**

 The local storage uses the local Storage object to store data for your entire website on a permanent basis. That means the **stored local data will be available on the next day, the next week, or the next year unless you remove it**.

* **localStorage.setItem(key, value)** stores the value associated with a key.
* **localStorage.getItem(key)** retrieves the value associated with the key.

// Check if the localStorage object exists

if(localStorage) {

// Store data localStorage.setItem("first\_name", "Peter");

// Retrieve data alert("Hi, " + localStorage.getItem("first\_name"));

}

else { alert("Sorry, your browser do not support local storage.");

}

var personObject = { name: "Peter", age: 18, married: false };

// Convert the person object into JSON string and save it into storage localStorage.setItem("personObject", JSON.stringify(personObject));

// Retrieve the JSON string

var jsonString = localStorage.getItem("personObject");

// Parse the JSON string back to JS object

var retrievedObject = JSON.parse(jsonString);

console.log(retrievedObject); // Accessing individual values console.log(retrievedObject.name); // Prints: Peter

console.log(retrievedObject.age); // Prints: 18

console.log(retrievedObject.married); // Prints: false

**Session storage**

## The session storage uses the sessionStorage object to store data on a temporary basis, for a single browser window or tab. The data disappears when session ends i.e. when the user closes that browser window or tab.

## // Check if the sessionStorage object exists

## if(sessionStorage) {

## // Store data sessionStorage.setItem("last\_name", "Parker");

## // Retrieve data alert("Hi, " + localStorage.getItem("first\_name") + " " + sessionStorage.getItem("last\_name"));

## } else {

## alert("Sorry, your browser do not support session storage.");

## }

## Application Cache

* **Offline browsing** — Users can use the application even when they're offline or there are unexpected disruptions in the network connection.
* **Improve performance** — Cached resources load directly from the user's machine rather than the remote server hence web pages load faster and performing better.
* **Reduce HTTP request and server load** — The browser will only have to download the updated/changed resources from the remote server that minimize the HTTP requests and saves precious bandwidth as well as reduce the load on the web server.
* <html lang="en" manifest="example.appcache">

**CACHE MANIFEST**

# v1.0 : 10-08-2014

**CACHE:**

# pages

index.html

# styles & scripts

css/theme.css

js/jquery.min.js

js/default.js

# images

/favicon.ico

images/logo.png

**NETWORK:**

login.php

**FALLBACK:**

// offline.html

## Create a Web Worker File

The simplest use of web workers is for **performing a time-consuming task**.

web worker that is specifically designed to do background work independently of other user-interface scripts, without affecting the performance of the page.

// Set up global variable

var worker;

function startWorker() {

// Initialize web worker

worker = new Worker("worker.js");

// Run update function, when we get a message from worker

worker.onmessage = update;

// Tell worker to get started worker.postMessage("start");

}

function update(event) {

// Update the page with current message from worker document.getElementById("result").innerHTML = event.data;

}

function stopWorker() {

// Stop the worker worker.terminate();

}

## Working with Timers

## 

## setTimeout(function, milliseconds)

## Execute a function or specified piece of code just once after a certain period of time,

## Executing code after a delay

## function myFunction() {

## alert('Hello World!');

## }

## <button onclick="setTimeout(myFunction, 2000)">Click Me</button>

## setInterval(function, milliseconds).

## Execute a function or specified piece of code repeatedly at fixed time intervals.

## Executing code at Regular Intervals

## var timeoutID;

## function showAlert() { alert('This is a JavaScript alert box.'); }

## function delayedAlert() {

## timeoutID = setTimeout(showAlert, 2000);

## }

## Clearing a timer can be done . Stopping code execution or cancelling a Timer

## clearTimeout(function, milliseconds)

## clearInterval(function, milliseconds).

## var timeoutID;

## function delayedAlert() {

## timeoutID = setTimeout(showAlert, 2000);

## }

## function showAlert() {

## alert('This is a JavaScript alert box.');

## }

## function clearAlert() {

## clearTimeout(timeoutID);

## }

## function stopClock() {

## clearInterval(intervalID);

## }

**ECMAScript 6 (or ES6)**

New features such as, **block-scoped variables**, **new loop for iterating over arrays and objects**, **template literals**

**1. block-scoped Variables :**

Let 🡪  if you declare a variable with the let keyword **inside a loop**, it does not exist outside of the loop(console.log = undefined)

for(let i = 0; i < 5; i++) {

console.log(i); // 0,1,2,3,4

}

console.log(i); // undefined

const  🡪 Constants are **read-only**, you **cannot reassign new values** to them and you can still change **object properties or array** elements

const PI = 3.14;

console.log(PI); // 3.14

PI = 10; // error

for...of 🡪  loops over **iterable objects such as arrays, strings**, etc..  loop is executed for each element of the iterable object

// Iterating over array

let letters = ["a", "b", "c", "d", "e", "f"];

for(let letter of letters) {

console.log(letter); // a,b,c,d,e,f

}

// Iterating over string

let greet = "Hello World!";

for(let character of greet) {

console.log(character); // H,e,l,l,o, ,W,o,r,l,d,!

}

for...in 🡪  loops through the **properties of an object**. If you want to **iterate over the properties of an object** you can use

## 2.Template Literals

## back-tick (` `) (grave accent)

## // Simple multi-line string

## let str = `The quick brown fox jumps over the lazy dog.`; // String with embedded variables and expression

## let a = 10; let b = 20;

## let result = `The sum of ${a} and ${b} is ${a+b}.`; console.log(result); // The sum of 10 and 20 is 30.

## 3. Default Values for Function Parameters

## function when it is called these default parameters values will be used

## function sayHello(name='World') {

## return `Hello ${name}!`;

## }

## console.log(sayHello()); // Hello World!

## console.log(sayHello('John')); // Hello John!

## 4.Arrow Functions

## [function expressions](https://www.tutorialrepublic.com/javascript-tutorial/javascript-functions.php#function-expressions) by opting out the function and return keywords. the fat arrow (=>) notation.

## // Function Expression

## var sum = function(a, b) { return a + b; } console.log(sum(2, 3)); // 5

## // Arrow function

## var sum = (a, b) => a + b; console.log(sum(2, 3)); // 5

## 5. this  🡪 the current execution context of a function

function Person(nickname, country) {

this.nickname = nickname;

this.country = country;

// Function Expression

this.getInfo = function() {

// Outer function context (Person object)

return () => {

// Inner function context (Person object)

alert(this.constructor.name); // Person

alert(`Hi, I'm ${this.nickname} from ${this.country}`);

};

}

}

let p = new Person('Rick', 'Argentina');

let printInfo = p.getInfo();

printInfo(); // Hi, I'm Rick from Argentina

## Classes

**ES6 classes make it easier to create objects, implement inheritance by using the extends keyword, and reuse the code.**

**class** 🡪 Class have *Class constructor and Class method.*  To create objects, implement inheritance by using the extends keyword, and reuse the code. This keyword followed by a class-name. Group of statements.

**ClassName** 🡪 (i.e. capitalizing the first letter of each word).

**Extends** 🡪 *Child class inherits from Parent*, Classes that inherit from other classes are referred to as derived classes or child classes

**super**() 🡪  *Call parent's constructor* and Square class inherits from the Parent class

**constructor**() 🡪 method is a special method for *creating and initializing an object* created with a **class**

class Rectangle {

// Class constructor

constructor(length, width) {

this.length = length;

this.width = width;

}

// Class method

getArea() {

return this.length \* this.width;

} }

// Square class inherits from the Rectangle class

class Square extends Rectangle {

// Child class constructor

constructor(length) {

// Call parent's constructor

super(length, length);

}

// Child class method

getPerimeter() {

return 2 \* (this.length + this.width);

} }

let rectangle = new Rectangle(5, 10);

alert(rectangle.getArea()); // 50

let square = new Square(5);

alert(square.getArea()); // 25

alert(square.getPerimeter()); // 20

alert(square instanceof Square); // true

alert(square instanceof Rectangle); // true

alert(rectangle instanceof Square); // false

## Modules

## module, in which each module is represented by a separate .js file. you can use the export or import statement in a module to export or import variables, functions, classes or any other entity to from other modules or files.

## <script type="module" src="app.js"></script>

## main.js

## let greet = "Hello World!";

## const PI = 3.14;

## function multiplyNumbers(a, b) { return a \* b; } // Exporting variables and functions

## export { greet, PI, multiplyNumbers };

## app.js

## import { greet, PI, multiplyNumbers } from './main.js';

## alert(greet); // Hello World!

## alert(PI); // 3.14

## alert(multiplyNumbers(6, 15)); // 90

A module is a place where we can group components, directives, services, and pipes. Module decides whether the components, directives, etc can be used by other modules, by exporting or hiding these elements. Every module is defined with a @NgModule decorator.  
By default, modules are of two types:

 Root Module

 Feature Module

Every application can have only one root module whereas, it can have one or more feature modules.  
A root module imports **BrowserModule**, whereas a feature module imports **CommonModule**.  
  
In the application that we created before, one can see that the root module is defined inside **app.module.ts** and this is how it looks:

import { BrowserModule } from '@angular/platform-browser';

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

import { AppComponent } from './app.component';

import { TestComponent } from './test/text.component';

@NgModule({

declarations: [

AppComponent,

TestComponent

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

## The Rest Parameters

## A rest parameter is specified by prefixing a named parameter with rest operator (...) i.e. three dots

## function sortNames(...names) { return names.sort(); } alert(sortNames("Sarah", "Harry", "Peter")); // Harry,Peter,Sarah

## function myFunction(a, b, ...args) { return args; } alert(myFunction(1, 2, 3, 4, 5)); // 3,4,5

## The Spread Operator

## The spread operator, which is also denoted by (...), performs the exact opposite function of the rest operator.

## function addNumbers(a, b, c) {

## return a + b + c;

## }

## let numbers = [5, 12, 8]; // ES5 way of passing array as an argument of a function

## alert(addNumbers.apply(null, numbers)); // 25

## // ES6 spread operator

## alert(addNumbers(...numbers)); // 25

## Destructuring Assignment

## To get an individual value of an array

## let fruits = ["Apple", "Banana"];

## let [a, b] = fruits; // Array destructuring assignment

## alert(a); // Apple

## alert(b); // Banana

## let person = {name: "Peter", age: 28};

## let {name, age} = person; // Object destructuring assignment

## alert(name); // Peter

## alert(age); // 28

## @Input()

## @Input() id: string | undefined;

## @Input() Parent to child component importing

## <templates>

## <pagination-controls (pageChange)="currentpg = $event"></pagination-controls>

## Angular JS

| **Angular Version** | **Date** | **Description** |
| --- | --- | --- |
| Angular 2 | 14.09.2016 | Initial Version of Angular |
| Angular 4 | 23.03.2017 | Version 4 |
| Angular 5 | 11.11.2017 | Version 5 |
| Angular 6 | 03-05-2018 | Version 6 |
| Angular 7 | 18-10-2018 | Version 7 |
| Angular 8 | 25-08-2019 | Version 8 |
| Angular 9 | 06-02-2020 | Version 9 |
| Angular 10 | 24-06-2020 | Version 10 |
| Angular 10.0.12 | 24-08-2020 | Version 10.0.12 |

## Install with node commond

## npm install -g @angular/cli

## ng new my-dream-app

## cd my-dream-app

## ng serve

## Commands

## *ng g c componentname*

## *ng g s servicename*

## *npm i bootstrap*

## *npm install jquery*

## *ng build –prod*

## *node server.js*

## cntrl 🡪 space (import line)

## routeLink

## Angular.json 🡪 main.ts 🡪 app.module.ts

* "main": "src/main.ts",
* platformBrowserDynamic().bootstrapModule(AppModule)
* bootstrap: [AppComponent]
* *Just-in-Time* (JIT), compiles your app in the browser at runtime.

## ****just-in-time**** (****JIT****) ****compilation**** (also **dynamic translation** or **run-time compilations**) is a way of executing [computer code](https://en.wikipedia.org/wiki/Computer_code) that involves [compilation](https://en.wikipedia.org/wiki/Compiler) during execution of a program — at [run time](https://en.wikipedia.org/wiki/Run_time_(program_lifecycle_phase)) — rather than prior to execution.

* *Ahead-of-Time* (AOT), compiles your app at build time on the server.

## An ahead-of-time (AOT) compiler converts your code during the build time before the browser downloads and runs that code.

## ng build --prod to build source code bundles which includes assets, JS files (main, vendor, and polyfills), index.html, and CSS. In this step, Angular uses the [Angular compiler](https://www.npmjs.com/package/@angular/compiler-cli) to build source code and they do it in [3 phases](https://angular.io/guide/aot-compiler#compilation-phases) which are code analysis, code generation, and template type checking. In this step, the bundle size will be smaller than bundle size when we build by JIT mode.

## **#1**Interpolation: **{{value}}** **#2**Propertybinding: **[property]=”value”** **#3**Eventbinding: **(event)=”function”** **#4** Two-way data binding: **[(ngModel)]=”value”**

## pristine: This property returns true if the element's contents have not been changed.

## dirty: This property returns true if the element's contents have been changed.

## untouched: This property returns true if the user has not visited the element.

## Services

**Services** Services are objects which get instantiated only once during the lifetime of an application. The main objective of a service is to share data, functions with different components of an Angular application.  
A service is defined using a **@Injectable** decorator. A function defined inside a service can be invoked from any component or directive.  
  
To create a service, run the following command:

ng g s test-service

## Directives

A directive is a class in Angular that is declared with a **@Directive** decorator.  
Every directive has its own behaviour and can be imported into various components of an application.  
  
**When to use a directive?**  
Consider an application, where multiple components need to have similar functionalities. The norm thing to do is by adding this functionality individually to every component but, this task is tedious to perform. In such a situation, one can create a **directive** having the required functionality and then, import the directive to components which require this functionality.  
  
**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 behaviour 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:

**onInit**

Angular has initialized all data-bound properties of a directive.

[**NgModule**](https://angular.io/api/core/NgModule)

The @[NgModule](https://angular.io/api/core/NgModule) decorator identifies **AppModule** as an [**NgModule**](https://angular.io/api/core/NgModule) class.

@[**NgModule**](https://angular.io/api/core/NgModule) takes a metadata object that tells Angular how to compile and launch the application.

* **declarations**—this application's lone component.
* **imports**—import [*BrowserModule*](https://angular.io/api/platform-browser/BrowserModule) to have browser specific services such as DOM rendering, sanitization, and location.
* **providers**—the service providers.
* **bootstrap**—the root component that Angular creates and inserts into the index.html host web page.

## Login Form Process

## FormBuilder initialization in constructor

## With expression function initialize form fields from FormBuilder in ngOnInit

## onSubmit Button get form values in Object and this Object calling in Services response from Services.ts file

## FormsModule and ReactiveFormsModule

1. **Template-driven** forms make use of the "**FormsModule(asynchronous**)", while **Reactive forms** are based on "**ReactiveFormsModule(synchronous**)".
2. **Template-driven** forms are **asynchronous** in nature, whereas **Reactive forms** are mostly **synchronous**.
3. In a template-driven approach, most of the logic is driven from the template, whereas in reactive-driven approach, the logic resides mainly in the component or typescript code. Let us get started by generating a component and then we'll update our form code.
4. [***https://www.pluralsight.com/guides/difference-between-template-driven-and-reactive-forms-angular***](https://www.pluralsight.com/guides/difference-between-template-driven-and-reactive-forms-angular)
5. [***https://www.cloudhadoop.com/2018/08/typescript-how-to-convert-object-to.html***](https://www.cloudhadoop.com/2018/08/typescript-how-to-convert-object-to.html)
6. [***https://blog.angular-university.io/introduction-to-angular-2-forms-template-driven-vs-model-driven/***](https://blog.angular-university.io/introduction-to-angular-2-forms-template-driven-vs-model-driven/)
7. [***https://coryrylan.com/blog/angular-form-builder-and-validation-management***](https://coryrylan.com/blog/angular-form-builder-and-validation-management)
8. [***https://www.pluralsight.com/guides/how-to-display-validation-messages-using-angular***](https://www.pluralsight.com/guides/how-to-display-validation-messages-using-angular)

## FormsModule

## Template – driven forms

## In Module.ts.file

## FomsModule (adding)

## In Add Form tag

## (ngSubmit)="onSubmit(f)" #f="ngForm"

## In TS file

## @ViewChild('f') courseForm: NgForm;

## Html file

## <input type="text" id="courseName" class="form-control" name="courseName" ngModel required #courseName="ngModel">

## Validations

## <div style="color:red" \*ngIf="courseName.errors && (courseName.dirty || courseName.touched)">

## <p \*ngIf="courseName.errors.required"> Course Name is required </p>

## </div>

## fb: FormBuilder

## The [FormBuilder](https://angular.io/api/forms/FormBuilder) provides syntactic sugar that shortens creating instances of a [FormControl](https://angular.io/api/forms/FormControl), [FormGroup](https://angular.io/api/forms/FormGroup), or [FormArray](https://angular.io/api/forms/FormArray)

## Group() 🡪 Construct a new [FormGroup](https://angular.io/api/forms/FormGroup) instance ( this.fb.group )

## Control() 🡪 Construct a new [FormControl](https://angular.io/api/forms/FormControl) with the given state, validators and options.

## Array() 🡪 Constructs a new [FormArray](https://angular.io/api/forms/FormArray) from the given array of configurations, validators and options.

## Modules Added In Application

## BrowserModule (<https://angular.io/api/platform-browser/BrowserModule>)

browser specific services such as DOM rendering, sanitization, and location.

## NgModule (<https://angular.io/guide/ngmodules>)

## configure the injector and the compiler and help organize related things together. An NgModule describes how the application parts fit together.

## HttpClientModule (https://angular.io/api/common/http/HttpClientModule)

## Configures the [dependency injector](https://angular.io/guide/glossary#injector) for [HttpClient](https://angular.io/api/common/http/HttpClient) with supporting services for XSRF

## AppRoutingModule (<https://angular.io/tutorial/toh-pt5>)

## Routes tell the Router which view to display when a user clicks a link or pastes a URL into the browser address bar.

## AppComponent (<https://angular.io/guide/bootstrapping>)

## The default application created by the Angular CLI only has one component, AppComponent, so it is in both the declarations and the bootstrap arrays.

## Router (<https://angular.io/guide/router>)

## The [Router](https://angular.io/api/router/Router) enables navigation by interpreting a browser URL as an instruction to change the view.

## Component, OnInit, ViewChild, AfterViewInit (https://angular.io/guide/lifecycle-hooks)

## The lifecycle continues with change detection, as Angular checks to see when data-bound properties change, and updates both the view and the component instance as needed

## empty, from

## FormBuilder, FormGroup, Validators (<https://angular.io/api/forms/FormBuilder>)

## The [FormBuilder](https://angular.io/api/forms/FormBuilder) provides syntactic sugar that shortens creating instances of a [FormControl](https://angular.io/api/forms/FormControl), [FormGroup](https://angular.io/api/forms/FormGroup), or [FormArray](https://angular.io/api/forms/FormArray). It reduces the amount of boilerplate needed to build complex forms.

## ActivatedRoute (https://angular.io/api/router/ActivatedRoute)

## Provides access to information about a route associated with a component that is loaded in an outlet. Use to traverse the [RouterState](https://angular.io/api/router/RouterState) tree and extract information from nodes.

## HttpClient (<https://angular.io/guide/http>)

## Most front-end applications need to communicate with a server over the HTTP protocol, in order to download or upload data and access other back-end services.

## Injectable (<https://angular.io/api/core/Injectable>)

## Decorator that marks a class as available to be provided and injected as a dependency.

## Environment

## An Angular Application Environment is JSON configuration information that tells the build system which files to change when you use ng build and ng serve

## 

## app.modules

## import { BrowserModule } from '@angular/platform-browser';

## import { NgModule } from '@angular/core';

## import { FormsModule, ReactiveFormsModule} from '@angular/forms';

## import { HttpClientModule } from '@angular/common/http';

## import { AppRoutingModule } from './app-routing.module';

## import { AppComponent } from './app.component';

## User Services modules/component

## import { HttpClient } from '@angular/common/http';

## import { Injectable } from '@angular/core';

## import { environment } from '../environments/environment';

## *private httpclient:HttpClient*

## *return this.httpclient.post(environment.baseUrl, RegObj);*

## *return this.httpclient.get('https://www.w3schools.com/angular/customers.php');*

## *userSignup(RegObj: any) {*

## *const contentHeaders = new HttpHeaders();*

## *contentHeaders.append('Authorization', 'Your token used in app');*

## *contentHeaders.append('Content-Type', 'application/json');*

## *contentHeaders.append('Access-Control-Allow-Origin', 'http://localhost:4200');*

## *return this.httpclient.post('http://localhost:8080/api/auth/signup', RegObj, { headers: contentHeaders });*

## *}*

## Login.modules/component

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

## import { FormBuilder, FormGroup, Validators } from '@angular/forms';

## import { empty, from } from 'rxjs';

## import { UserService } from '../user.service';

## import { EditdataComponent } from '../editdata/editdata.component';

## import { Router } from '@angular/router';

## import { ActivatedRoute } from '@angular/router'; //For Updated module

## *private fb: FormBuilder, private uservice: UserService, private router: Router, private route: ActivatedRoute*

## *this.loginForm = this.fb.group({ })*

## *this.uservice.getUserDetails().subscribe((response: any) => {*

## *this.responsedata = response.records;*

## *this.obj1.push(this.responsedata);*

## *}) }*

## *localStorage.setItem('localStorage-editValue', JSON.stringify(editValue));*

## *this.router.navigate(['/editdata', editValue.Name]);*

## *let index = this.responsedata.findIndex(x => x.Name === updatedData);*

## Signup.modules/component

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

## import { FormBuilder, FormGroup, Validators } from '@angular/forms';

## import { UserService } from '../user.service';

## *private fb:FormBuilder, private uservice:UserService*

## *localStorage.setItem('userObj', JSON.stringify(RegObj));*

## *let getLocal = JSON.parse(localStorage.getItem('userObj'));*

## *for(let i=0; i<=getLocal.length; i++){*

## *this.localArray.push(this.getLocal[i]);*

## *}*

## *throw new Error('Method not implemented.');*

## editdata.modules/component

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

## import { FormBuilder, FormGroup, Validators } from '@angular/forms';

## import { ActivatedRoute } from '@angular/router';

## import { SharedService } from '../shared.service';

## import { UserService } from '../user.service';

## *private fb: FormBuilder, private route: ActivatedRoute, private uservice: UserService, private share: SharedService,*

## Method : 1

## *let Name = this.route.snapshot.paramMap.get('name');*

## *this.uservice.getUserDetails().subscribe((response: any) => { …})*

## *this.responsedata.filter((filterData: any) => { ….})*

## *this.editDataForm.patchValue({…})*

## *this.responsedata.filter((filterData: any) => {*

## *if (filterData.Name == updatedData.Name) {*

## *let index = this.responsedata.findIndex(x => x.Name ===updatedData.Name);*

## *this.responsedata.splice(index,1, updatedData);*

## *}*

## *})*

## Method : 2

## *this.userId = this.route.snapshot.paramMap.get('id');*

## *console.log('ID', this.userId);*

## *if (this.userId) {*

## *debugger;*

## *this.getUserById(this.userId)*

## *}*

## *else {*

## *this.toaster.error('id not available');*

## *}*

## *}*

## *getUserById(userId: any) {*

## *let userIdObj = {*

## *'userId': userId*

## *}*

## *this.userserve.getUsersDetailsById(userIdObj).subscribe((response: any) => {*

## *this.responseData = response.results;*

## *this.Location = this.responseData.Location;*

## *console.log('this.responsedata...', this.responseData);*

## *this.editDataForm.patchValue({*

## *'username': this.responseData.username,*

## *'email': this.responseData.email,*

## *'address': this.responseData.address,*

## *'pincode': this.responseData.pincode*

## *})*

## *})*

## *}*

## Get users-details

## *this.userserve.getUsersDetails().subscribe((response:any)=>{*

## *this.responsedata = response.result;*

## *console.log('get users data', this.responsedata);*

## *});*

## Get users-details

## *userEdit(id: any) {*

## *console.log('user Id', id);*

## *localStorage.setItem('localStorage-editValue', JSON.stringify(id));*

## *this.router.navigate(['/userEdit', id]);*

## *}*

## Delete User Row

## *deleteDataRow(updatedData) {*

## *debugger*

## *console.log('updatedData...', updatedData);*

## *let index = this.responsedata.findIndex(x => x.Name === updatedData);*

## *this.obj1=[];*

## *this.responsedata.slice(index);*

## *this.obj1.push(this.responsedata);*

## *}*

## Get Dashboard-details (From Login)

## *this.userData = localStorage.getItem('userObj');*

## *this.parsedata = JSON.parse(this.userData);*

## *console.log('retrievedObject: ', this.parsedata);*

## File Image upload

## *imageUpload(event: any) {*

## *const file = event.target.files && event.target.files[0];*

## *if (file) {*

## *let formdata = new FormData();*

## *formdata.append('file', file);*

## *this.userserve.userImage(formdata).subscribe((response: any) => {*

## *this.responsedata = response;*

## *this.toaster.success(this.responsedata);*

## *this.Key = response.data.key;*

## *this.Location = response.data.Location;*

## *}, (error: any) => {*

## *alert(error.message);*

## *})*

## *}*

## *else {*

## *return;*

## *}*

## *}*

## Disabled button with Checkbox

## *checked() {*

## *let check = this.signup.controls['checkboxTerms'].value*

## *console.log('value..', check);*

## *if (check) {*

## *this.checkedBtn = false;*

## *}*

## *else {*

## *this.checkedBtn = true;*

## *}*

## *}*

## Toaster messages Script

## *if (this.responsedata.status == 200) {*

## *this.toaster.success(this.responsedata.message);*

## *}*

## *if (this.responsedata.status == 400) {*

## *this.toaster.error(this.responsedata.message);*

## *}*

## Routing Script

## this.router.navigate(['/login'])

## Common Script

## this.obj1=[];

## this.responsedata.slice(index);

## this.obj1.push(this.responsedata);

## Sign up form Coding and SharedSerices style:

## import { SharedService } from '../shared.service';

## 

## ngOnInit(): void {

## this.signup = this.fb.group({ 'username': ['', Validators.required],})

## }

## onSubmit(data: any) {

## let RegObj = { 'username': this.signup.controls['username'].value, }

## this.userserve.userSignup(RegObj).subscribe((response: any) => {

## this.responsedata = response;

## if (this.responsedata.status == 200) {

## this.loading = false;

## this.toaster.success(this.responsedata.message);

## this.signup.reset();

## this.router.navigate(['/login'])

## }

## if (this.responsedata.status == 400) {

## this.loading = false;

## this.toaster.error(this.responsedata.message);

## }

## }, (error: any) => {

## if (error) {

## this.loading = false;

## this.toaster.error(

## error.message)

## }

## })

## }

## }

## Sign up Edit form Coding and SharedSerices style:

## import { SharedService } from '../shared.service';

## 

## ngOnInit(): void {

## this.signup = this.fb.group({ 'username': ['', Validators.required], });

## this.userId = this.route.snapshot.paramMap.get('id');

## if (this.userId) {

## this.getUserById(this.userId)

## }

## else {

## this.toaster.error('id not available');

## }

## }

## getUserById(userId: any) {

## let userIdObj = {

## 'userId': userId

## }

## this.userserve.getUsersDetailsById(userIdObj).subscribe((response: any) => {

## this.responseData = response.results;

## this.Location = this.responseData.Location;

## this.editDataForm.patchValue({

## 'username': this.responseData.username,

## 'email': this.responseData.email,

## 'address': this.responseData.address,

## 'pincode': this.responseData.pincode

## })

## })

## }

## onSubmit(data: any) {

## let RegObj = { 'username': this.signup.controls['username'].value, }

## this.userserve.editUser(RegObj).subscribe((response: any) => {

## this.responseData = response;

## if (this.responseData.status == 200) {

## this.toaster.success(this.responseData.results);

## this.router.navigate(['/userDetails']);

## }

## else {

## this.toaster.error(this.responseData.results);

## }

## });

## }

## Update Data SharedSerices style:

## import { SharedService } from '../shared.service';

## 

## productEdit(productRow:any){

## // debugger;

## this.DataShare.addProductEdit.next(productRow);

## this.router.navigate(['/addProducts/', productRow.Id]);

## }

## this.share.addProductEdit.subscribe((response: any) => {

## this.responsedata = response;

## this.Location = this.responsedata.product\_Url;

## this.addProducts.patchValue({

## 'productName': this.responsedata.productname,

## 'productBrand': this.responsedata.brand,

## 'productQty': this.responsedata.quantity,

## 'productQtyType': this.responsedata.quantityType,

## 'productPrice': this.responsedata.price,

## 'productDesc': this.responsedata.productDesc,

## 'categoryList': this.responsedata.productCategory,

## });

## })

## this.onOptionsSelected(this.responsedata.quantityType);

## this.productId = this.Activatedroute.snapshot.paramMap.get('productId');

## {path:'addProducts/:productId', component:AddProductsComponent} ,app-routing.module.ts 🡪 productId

## Search Data:

## <div class="col-sm-5 search\_input">

## <input type="text" class="form-control search\_input" #search name="productname" [(ngModel)]="productname"

## (ngModelChange)="Search(search.value)" placeholder="Enter product name to search">

## <button type="button" routerLink="/addProducts" class="btn btn-primary">

## Add Product

## </button>

## </div>

## serarchProduct(sdata: any) {

## return this.httpclient.post(environment.BaseUrl + 'searchAllProducts', sdata);

## }

## Search(data: any) {

## if (data.length != 0) {

## let sdata = {

## productName: data

## }

## this.\_ps.serarchProduct(sdata).subscribe((response: any) => {

## this.responsedata = response.result;

## });

## }

## else {

## this.getProduct();

## }

## }

## Custom Table and Sorting, Pagination:

## <table class="table table-striped">

## <thead>

## <tr>

## <th scope="col" (click)="sort('Id')">User ID &nbsp;<i class="fa fa-sort"></i></th>

## <th scope="col">Product Image</th>

## <th scope="col" (click)="sort('productname')">Product Name &nbsp;<i class="fa fa-sort"></i></th>

## <th scope="col" (click)="sort('brand')">Brand &nbsp;<i class="fa fa-sort"></i></th>

## <th scope="col" (click)="sort('quantityType')">Qty Type &nbsp;<i class="fa fa-sort"></i></th>

## <th scope="col" (click)="sort('quantity')">Qty &nbsp;<i class="fa fa-sort"></i></th>

## <th scope="col" (click)="sort('price')">Price &nbsp;<i class="fa fa-sort"></i></th>

## <th scope="col">Action</th>

## </tr>

## </thead>

## <tbody>

## <tr

## \*ngFor="let item of responsedata | orderBy:key:reverse | paginate:{itemsPerPage: pageNumber,currentPage:currentpg} let i = index;">

## <td>{{item?.Id}}</td>

## <td><img [src]="item.product\_Url!= null ? item.product\_Url : product\_image"

## class="user\_img m-auto" alt="User Image"></td>

## <td>{{item?.productname}}</td>

## <td>{{item?.brand}}</td>

## <td>{{item?.quantityType}}</td>

## <td>{{item?.quantity}}</td>

## <td>{{item?.price}}</td>

## <td><a (click)="productEdit(item)"><i class="fa fa-edit mr-2"></i></a>

## <a (click)="productDelete(item.Id)"> <i class="fa fa-trash"></i></a>

## </td>

## </tr>

## </tbody>

## </table>

## <div class="paginate">

## <button class="btn btn-plus ml-3" (click)="dec()"><i class="fa fa-minus"></i></button>

## <input class="form-control pageItems" [value]="pageNumber">

## <button class="btn btn-plus" (click)="inc()"><i class="fa fa-plus"></i></button>

## <span class="m-1 mr-auto"># of Items per Page</span>

## <pagination-controls (pageChange)="currentpg = $event"></pagination-controls>

## 

## </div>

## currentpg: any;

## pageNumber: number = 5;

## productname: any;

## responsedata: any = [];

## p: number = 1;

## pageSize = 8;

## products: Product[] = []

## config: any;

## prodctId: any;

## onPageChange(event: any) {

## this.config.currentPage = event;

## }

## key: string = 'Id';

## reverse: boolean = false;

## sort(key: any) {

## this.key = key;

## this.reverse = !this.reverse;

## }

## pageChanged(event: any) {

## this.config.currentPage = event;

## }

## dec() {

## if (this.pageNumber < this.responsedata.length) { this.pageNumber -= 2; }

## }

## inc() {

## this.pageNumber += 2;

## if (this.pageNumber > 1) { }

## }

## Select Field options:

## numCountText1 = ['100g', '200g', '250g', '300g', '400g', '500g', '1000g']

## 'numCountTextG': this.fb.array(this.numCountText1),

## <div class="form-check form-check-inline"

## \*ngFor="let numCountTextG of numCountText1; let i = index"

## formArrayName="numCountTextG">

## <input type="checkbox" [formControlName]="i" class="mr-2">

## <label class="form-check-label">{{numCountTextG}}</label>

## </div>

## Map and Push array:

## public getProductQt(valueLIst: any) {

## this.gramsList = [];

## valueLIst.map((x: any) => { if (typeof (x) == 'string') { this.gramsList.push(x) } });

## }

## Slice :

## getQty(priceObj: any, \_mg: any) {

## debugger;

## if (priceObj.quantityType == 'kgs') {

## let qt = Number(\_mg.slice(0, \_mg.length - 2))

## priceObj.QtyPrice = Number(priceObj.price) \* qt;

## }

## }

## breakpoints: (

## xs: 0,

## sm: 576px,

## md: 768px,

## lg: 992px,

## xl: 1200px,

## xxl: 1400px

## );

## HOST: "localhost",

## USER: "root",

## PASSWORD: "Root@12345",

## git status

## git add .

## git commit -m "Comments"

## git checkout -b "Login-Dashaboard-added"

## git push origin Login-Dashaboard-added

sendMessage

OtpVerification

## Product Image

## Product Type

## Product name

## Quantity (50g, 100g, 250g, 500g, 1000g )

## Quantity Type (gram, Litre, Number)

## Price (50g, 100g, 250g, 500g, 1000g )

## Interview Question

## 1. Why were client-side frameworks like Angular introduced?

## Back in the day, web developers used VanillaJS and jQuery to develop dynamic websites but, as the logic of one's website grew, the code became more and more tedious to maintain. For applications that use complex logic, developers had to put in extra effort to maintain separation of concerns for the app. Also, jQuery did not provide facilities for data handling across views. For tackling the above problems, client-side frameworks like Angular came into the picture, which made life easier for the developers by handling separation of concerns and dividing code into smaller bits of information (In the case of Angular, called Components). Client-side frameworks allow one to develop advanced web applications like Single-Page-Application. Not that we cannot develop SPAs using VanillaJS, but by doing so, the development process becomes slower.

## 2. How does an Angular application work?

## Every Angular app consists of a file named angular.json. This file will contain all the configurations of the app. While building the app, the builder looks at this file to find the entry point of the application. Following is an image of the angular.json file:

## 

## "build": {

## "builder": "@angular-devkit/build-angular:browser",

## "options": {

## "outputPath": "dist/angular-starter",

## "index": "src/index.html",

## "main": "src/main.ts",

## "polyfills": "src/polyfills.ts",

## "tsConfig": "tsconfig.app.json",

## "aot": false,

## "assets": [

## "src/favicon.ico",

## "src/assets"

## ],

## "styles": [

## "./node\_modules/@angular/material/prebuilt-themes/deeppurple-amber.css",

## "src/style.css"

## ]

## }

## }

## 

## 

## 

## Inside the build section, the main property of the options object defines the entry point of the application which in this case is main.ts. The main.ts file creates a browser environment for the application to run, and, along with this, it also calls a function called bootstrapModule, which bootstraps the application. These two steps are performed in the following order inside the main.ts file:

## import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

## 

## platformBrowserDynamic().bootstrapModule(AppModule)

## In the above line of code, AppModule is getting bootstrapped. The AppModule is declared in the app.module.ts file. This module contains declarations of all the components. Below is an example of app.module.ts file:

## 

## import { BrowserModule } from '@angular/platform-browser';

## import { NgModule } from '@angular/core';

## import { AppComponent } from './app.component';

## @NgModule({

## declarations: [

## AppComponent

## ],

## imports: [

## BrowserModule

## ],

## providers: [],

## entryComponents: [],

## bootstrap: [AppComponent]

## })

## export class AppModule { }

## 

## 

## 

## As one can see in the above file, AppComponent is getting bootstrapped. This component is defined in app.component.ts file. This file interacts with the webpage and serves data to it. Below is an example of app.component.ts file:

## 

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

## @Component({

## selector: 'app-root',

## templateUrl: './app.component.html',

## styleUrls: ['./app.component.css']

## })

## export class AppComponent {

## title = 'angular';

## }

## 

## 

## 

## Each component is declared with three properties: 1. Selector - used for accessing the component 2. Template/TemplateURL - contains HTML of the component 3. StylesURL - contains component-specific stylesheets After this, Angular calls the index.html file. This file consequently calls the root component that is app-root. The root component is defined in app.component.ts. This is how the index.html file looks:

## 

## <!doctype html>

## <html lang="en">

## <head>

## <meta charset="utf-8">

## <title>Angular</title>

## <base href="/">

## <meta name="viewport" content="width=device-width, initial-scale=1">

## </head>

## <body>

## <app-root></app-root>

## </body>

## </html>

## 

## 

## 

## The HTML template of the root component is displayed inside the <app-root> tags. This is how every angular application works.

## 3. What are some of the advantages of Angular over other frameworks?

##  Features that are provided out of the box - Angular provides a number of built-in features like,routing, state management, rxjs library and http servicesstraight out of the box. This means that one does not need tolook for the above stated features separately. They are allprovided with angular.

##  Declarative UI - Angular uses HTML to render the UI of an application. HTML isa declarative language and is much easier to use than JavaScript.

##  Long-term Google support - Google announced Long-term support for Angular. This means that Google plans to stick with Angular and further scale up its ecosystem.

## 4. List out differences between AngularJS and Angular

## Architecture

## AngularJS uses MVC or Model-View-Controller architecture, where the Model contains the business logic, Controller processes information and View shows the information present in the Model. Angular replaces controllers with Components. Components are nothing but directives with a predefined template.

## Language

## AngularJS uses JavaScript language, which is a dynamically typed language. Angular uses TypeScript language, which is a statically typed language and is a superset of JavaScript. By using statically typed language, Angular provides better performance while developing larger applications.

## Mobile Support

## AngularJS does not provide mobile support. Angular is supported by all popular mobile browsers.

## Structure

## While developing larger applications, the process of maintaining code becomes tedious in the case of AngularJS. In the case of Angular, it is easier to maintain code for larger applications as it provides a better structure.

## Expression Syntax

## While developing an AngularJS application, a developer needs to remember the correct ng-directive for binding an event, or a property. Whereas in Angular, property binding is done using "[ ]" attribute and event binding is done using "( )" attribute.

## 5. What is AOT compilation? What are the advantages of AOT?

## Every Angular application consists of components and templates which the browser cannot understand. Therefore, all the Angular applications need to be compiled first before running inside the browser. Angular provides two types of compilation:

##  JIT(Just-in-Time) compilation

##  AOT(Ahead-of-Time) compilation

## https://assets.interviewbit.com/assets/skill_interview_questions/angular/Angular-compilation-c5d30f9767e4b1a1447f35c680f0be0a81ad3131351419e4224d7e6f6b5d869f.png.gz

## In JIT compilation, the application compiles inside the browser during runtime. Whereas in the AOT compilation, the application compiles during the build time. The advantages of using AOT compilation are:

##  Since the application compiles before running inside the browser, the browser loads the executable code and renders the application immediately, which leads to faster rendering.

##  In AOT compilation, the compiler sends the external HTML and CSS files along with the application, eliminating separate AJAX requests for those source files, which leads to fewer ajax requests.

##  Developers can detect and handle errors during the building phase, which helps in minimizing errors.

##  The AOT compiler adds HTML and templates into the JS files before they run inside the browser. Due to this, there are no extra HTML files to be read, which provide better security to the application.

## By default, angular builds and serves the application using JIT compiler:

## ng build ng serve

## For using AOT compiler following changes should be made:

## ng build --aot ng serve --aot

## 6. Explain Components, Modules and Services in Angular

## For better understanding, I would like you to create an Angular application by running the following inside the command terminal:

## ng new angularApp

## 

## The above command will create an angular application in the directory. Next, let's move on to understand Components, Modules, and Services. Components In Angular, components are the basic building blocks, which control a part of the UI for any application. A component is defined using the @Component decorator. Every component consists of three parts, the template which loads the view for the component, a stylesheet which defines the look and feel for the component, and a class that contains the business logic for the component. For creating a component, inside the command terminal, navigate to the directory of the application created, and run the following command:

## ng generate component test

## 

## Or

## ng g c test

## 

## One can see the generated component inside src/app/test folder. The component will be defined inside test.component.ts and this is how it looks:

## 

## 

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

## @Component({

## selector: 'app-test',

## templateUrl: './test.component.html',

## styleUrls: ['./test.component.css']

## })

## export lass TestComponent implements OnInit {

## constructor() {}

## ngOnInit() {

## }

## }

## 

## 

## As we can see in the above image, our component is defined with @Component decorator.

## Modules A module is a place where we can group components, directives, services, and pipes. Module decides whether the components, directives, etc can be used by other modules, by exporting or hiding these elements. Every module is defined with a @NgModule decorator. By default, modules are of two types:

##  Root Module

##  Feature ModuleEvery application can have only one root module whereas, it can have one or more feature modules. A root module imports BrowserModule, whereas a feature module imports CommonModule. In the application that we created before, one can see that the root module is defined inside app.module.ts and this is how it looks:

## 

## import { BrowserModule } from '@angular/platform-browser';

## import { NgModule } from '@angular/core';

## import { AppComponent } from './app.component';

## import { TestComponent } from './test/text.component';

## @NgModule({

## declarations: [

## AppComponent,

## TestComponent

## ],

## imports: [

## BrowserModule

## ],

## providers: [],

## bootstrap: [AppComponent]

## })

## export class AppModule { }

## 

## 

## We can see in the above image that the component we created earlier is already imported in the declarations array. To create a feature module, run the following command:

## ng g m test-module

## The module is created inside the src/app/test-module/test-module.module.ts file:

## 

## import { NgModule } from '@angular/core';

## import { CommonModule } from '@angular/common';

## @NgModule({

## declarations: [],

## imports: [

## CommonModule

## ]

## })

## export class TestModuleModule { }

## 

## 

## As one can see, CommonModule is imported since this is a feature module.

## Services Services are objects which get instantiated only once during the lifetime of an application. The main objective of a service is to share data, functions with different components of an Angular application. A service is defined using a @Injectable decorator. A function defined inside a service can be invoked from any component or directive. To create a service, run the following command:

## ng g s test-service

## 

## The service will be created inside src/app/test-service.service.ts:

## 

## import { Injectable } from '@angular/core';

## @Injectable({

## providedIn: 'root'

## })

## export class TestServiceService {

## constructor() { }

## }

## 

## 

## Any method/function defined inside the TestServiceService class can be directly used inside any component by just importing the service.

## 7. What are lifecycle hooks in Angular? Explain a few lifecycle hooks.

## Every component in Angular has a lifecycle, different phases it goes through from the time of creation to the time it's destroyed. Angular provides hooks to tap into these phases and trigger changes at specific phases in a lifecycle.

## https://assets.interviewbit.com/assets/skill_interview_questions/angular/lifecycle-hooks-in-Angular-04aa5ba83e8ae72d56f84107defdde32783ce704089ba892b0217bb9c51ed9d8.png.gz

## 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. Let’s understand how to use ngOnInit hook, since it’s the most oftenly used hook. If one has to process lot of data during component creation, it’s better to do it inside ngOnInit hook rather than the constructor:

## 

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

## @Component({

## selector: 'app-test',

## templateUrl: './test.component.html',

## styleUrls: ['./test.component.css']

## })

## export class TestComponent implements OnInit {

## constructor() { }

## ngOnInit() {

## this.processData();

## }

## processData(){

## // Do something..

## }

## }

## 

## 

## As you can see we have imported OnInit but we have used ngOnInit function. This principle should be used with the rest of the hooks as well.

## 8. Explain string interpolation and property binding in Angular.

## String interpolation and property binding are parts of data-binding in Angular. Data-binding is a feature in angular, which provides a way to communicate between the component(Model) and its view(HTML template). Data-binding can be done in two ways, one-way binding and two-way binding. In Angular, data from the component can be inserted inside the HTML template. In one-way binding, any changes in the component will directly reflect inside the HTML template but, vice-versa is not possible. Whereas, it is possible in two-way binding. String interpolation and property binding allow only one-way data binding. String interpolation uses the double curly braces {{ }} to display data from the component. Angular automatically runs the expression written inside the curly braces, for example, {{ 2 + 2 }} will be evaluated by Angular and the output 4, will be displayed inside the HTML template. Using property binding, we can bind the DOM properties of an HTML element to a component's property. Property binding uses the square brackets [ ] syntax.

## 9. How are Angular expressions different from JavaScript expressions?

## The first and perhaps, the biggest difference is that Angular expressions allow us to write JavaScript in HTML which is not the case when it comes to JavaScript expressions. Next, Angular expressions are evaluated against a local scope object whereas JavaScript expressions against global window object. Let's understand that better with an example : Consider the following component named test:

## 

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

## @Component({

## selector: 'app-test',

## template: `

## <h4>{{message}}</h4>

## `,

## styleUrls: ['./test.component.css']

## })

## export class TestComponent implements OnInit {

## message:string = “Hello world”;

## constructor() { }

## ngOnInit() {

## }

## }

## 

## 

## As one can see that Angular expression is used to display message property of a component. Since we are using Angular expressions, in the present template, we cannot access a property outside of its local scope, which in this case is TestComponent. This proves that Angular expressions are always evaluated based on scope object rather than the global object. Next difference is how Angular expressions handle null and undefined. Consider the following JavaScript example:

## 

## <!DOCTYPE html>

## <html lang="en">

## <head>

## <meta charset="UTF-8">

## <meta name="viewport" content="width=device-width, initial-scale=1.0">

## <title>JavaScript Test</title>

## </head>

## <body>

## <div id="foo"><div>

## </body>

## <script>

## 'use strict';

## let bar = {};

## document.getElementById('foo').innerHTML = bar.x;

## </script>

## </html>

## 

## 

## If you run the above code, you will see undefined displayed on the screen. Although it’s not ideal to leave any property undefined, the user does not need to see this. Now consider the following Angular example:

## 

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

## @Component({

## selector: 'app-new',

## template: `

## <h4>{{message}}</h4>

## `,

## styleUrls: ['./new.component.css']

## })

## export class NewComponent implements OnInit {

## message:object = {};

## constructor() { }

## ngOnInit() {

## }

## }

## 

## 

## If you render the above component, you will not see undefined being displayed on the screen. Next, in Angular expressions one cannot use loops, conditionals and exceptions. The difference which makes Angular expressions quite beneficial is the use of pipes. Angular uses pipes(called filters in AngularJS), which can be used to format data before displaying it. Let’s see one predefined pipe in action:

## 

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

## @Component({

## selector: 'app-new',

## template: `

## <h4>{{message | lowercase}}</h4>

## `,

## styleUrls: ['./new.component.css']

## })

## export class NewComponent implements OnInit {

## message:string = "HELLO WORLD";

## constructor() { }

## ngOnInit() {

## }

## }

## 

## 

## In the above code we have used a predefined pipe called lowercase, which transforms all the letters in lowercase. Therefore, if you render the above component, you will see “hello world” being displayed. In contrast, JavaScript does not have the concept of pipes.

## 10. How are observables different from promises?

## The first difference is that an Observable is lazy whereas a Promise is eager.

|  |  |
| --- | --- |
| Promise | Observable |
| Emits a single value | Emits multiple values over a period of time |
| Not Lazy | Lazy. An observable is not called until we subscribe to the observable |
| Cannot be cancelled | Can be cancelled by using the unsubscribe() method |
|  | Observable provides operators like map, forEach, filter, reduce, retry, retryWhen etc. |

## Consider the following Observable:

## 

## const observable = rxjs.Observable.create(observer => {

## console.log('Text inside an observable');

## observer.next('Hello world!');

## observer.complete();

## });

## console.log('Before subscribing an Observable');

## observable.subscribe((message)=> console.log(message));

## 

## 

## When you run the above Observable, you can see messages being displayed in the following order:

## Before subscribing an Observable Text inside an observable Hello world!

## As you can see, observables are lazy. Observable runs only when someone subscribes to them hence, the message “Before subscribing…” is displayed ahead of the message inside the observable. Now let’s consider a Promise:

## 

## const promise = new Promise((resolve, reject) => {

## console.log('Text inside promise');

## resolve('Hello world!');

## });

## console.log('Before calling then method on Promise');

## greetingPoster.then(message => console.log(message));

## 

## 

## Running the above promise, the messages will be displayed in the following order:

## Text inside promise Before calling then method on Promise Hello world!

## As you can see the message inside Promise is displayed first. This means that a promise runs before the then method is called. Therefore, promises are eager. The next difference is that Promises are always asynchronous. Even when the promise is immediately resolved. Whereas an Observable, can be both synchronous and asynchronous. The above example of an observable is the case to show that an observable is synchronous. Let’s see the case where an observable can be asynchronous:

## 

## const observable = rxjs.Observable.create(observer => {

## setTimeout(()=>{

## observer.next('Hello world');

## observer.complete();

## },3000)

## });

## console.log('Before calling subscribe on an Observable');

## observable.subscribe((data)=> console.log(data));

## console.log('After calling subscribe on an Observable');

## 

## 

## The messages will be displayed in the following order:

## Before calling subscribe on an Observable After calling subscribe on an Observable Hello world!

## You can see in this case, observable runs asynchronously. The next difference is that Observables can emit multiple values whereas Promises can emit only one value. The biggest feature of using observables is the use of operators. We can use multiple operators on an observable whereas, there is no such feature in a promise.

## 11. Angular by default, uses client-side rendering for its applications.

## Can one make an angular application to render on the server-side? Yes, angular provides a technology called Angular Universal, which can be used to render applications on the server-side. The advantages of using Angular Universal are :

##  First time users can instantly see a view of the application. This benefits in providing better user experience.

##  Many search engines expect pages in plain HTML, thus, Universal can make sure that your content is available on every search engine, which leads to better SEO.

##  Any server-side rendered application loads faster since rendered pages are available to the browser sooner.

## 12. What are directives in Angular?

## A directive is a class in Angular that is declared with a @Directive decorator. Every directive has its own behaviour and can be imported into various components of an application. When to use a directive? Consider an application, where multiple components need to have similar functionalities. The norm thing to do is by adding this functionality individually to every component but, this task is tedious to perform. In such a situation, one can create a directive having the required functionality and then, import the directive to components which require this functionality. 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 behaviour 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>

## 

## 

## 13. How does one share data between components in Angular?

## Following are the commonly used methods by which one can pass data between components in angular:

## https://assets.interviewbit.com/assets/skill_interview_questions/angular/data-sharing-in-angular-6c40ab49398d4ea0fb449ca0979841c30ed4678759a9483d35fbecdaae93e0de.jpg.gz

## 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.

## https://assets.interviewbit.com/assets/skill_interview_questions/angular/data-sharing-angular-1b6dc6e218d9e630b66be934088703015d6d78af35eee69c6cc529ae895f51fd.png.gz

## 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. Child to parent 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;

## }

## 

## 

## 14. Explain the concept of Dependency Injection?

## Dependency injection is an application design pattern which is implemented by Angular. It also forms one of the core concepts of Angular. So what is dependency injection in simple terms? Let’s break it down, dependencies in angular are nothing but services which have a functionality. Functionality of a service, can be needed by various components and directives in an application. Angular provides a smooth mechanism by which we can inject these dependencies in our components and directives. So basically, we are just making dependencies which are injectable across all components of an application.

## https://assets.interviewbit.com/assets/skill_interview_questions/angular/Dependency-Injection-d0dee538b6ff0420395c8f85502e77a39ed80296f1699584bc8d056f9d70ce66.png.gz

## Let’s understand how DI (Dependency Injection) works: Consider the following service, which can be generated using:

## ng g service test

## 

## import { Injectable } from '@angular/core';

## @Injectable({

## providedIn: 'root'

## })

## export class TestService {

## importantValue:number = 42;

## constructor() { }

## returnImportantValue(){

## return this.importantValue;

## }

## }

## 

## 

## As one can notice, we can create injectable dependencies by adding the @Injectable decorator to a class. We inject the above dependency inside the following component:

## 

## import { TestService } from './../test.service';

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

## @Component({

## selector: 'app-test',

## templateUrl: './test.component.html',

## styleUrls: ['./test.component.css']

## })

## export class TestComponent implements OnInit {

## value:number;

## constructor(private testService:TestService) { }

## ngOnInit() {

## this.value = this.testService.returnImportantValue();

## }

## }

## 

## 

## One can see we have imported our TestService at the top of the page. Then, we have created an instance inside the constructor of the component and implemented the returnImportantValue function of the service. From the above example, we can observe how angular provides a smooth way to inject dependencies in any component.

## 15. Explain MVVM architecture

## MVVM architecture consists of three parts: 1. Model 2. View 3. ViewModel

## https://assets.interviewbit.com/assets/skill_interview_questions/angular/MVVM-architecture-4ff193d66d01ae6acf20f9c891ee657bbea63b52750572288a7c1b557ca0736a.png.gz

## Model contains the structure of an entity. In simple terms it contains data of an object. View is the visual layer of the application. It displays the data contained inside the Model. In angular terms, this will be the HTML template of a component. ViewModel is an abstract layer of the application. A viewmodel handles the logic of the application. It manages the data of a model and displays it in the view. View and ViewModel are connected with data-binding (two-way data-binding in this case). Any change in the view, the viewmodel takes a note and changes the appropriate data inside the model.

**Now, let's follow bellow step to creating crud app with angular 11.**

* Step 1: Create New App. ... ng new my-crud-app -–routing
* Step 2: Install Bootstrap. ... npm install bootstrap -–save
* Step 3: Create Post Module. ... ng generate module post –routing
* Step 4: Create Component For Module. ... ng generate component post/index etc
* Step 5: Create Route. ... { path: 'post', redirectTo: 'post/index', pathMatch: 'full'},
  + - { path: 'post/index', component: IndexComponent },
* Step 6: Create Interface. ... ng generate interface post/post
* Step 7: Create Services. ... ng generate service post/post
* Step 8: Update Component Logic and Template.