# Intro

Mongo (DB)

document oriented database written in C++

Collection -> Documents -> sets of name-value pairs

Node.js (environment)

Node.js is a server side cross-platform open source JavaScript execution environment

Express.js(back end which runs on Node)

Express.js is a lightweight Node.js based application used for back-end development

Angular (Front End)

Angular is an open source JavaScript framework developed by Google, for building both mobile and desktop web applications used for rapid front-end development

Samples: <https://mertjf.github.io/tailblocks/> <https://tailwindcss.com/components/cards>

URL (Uniform Resource Locator)

Protocol://Host:Port/Path

HTTP is a request/response based stateless and connectionless protocol.

stateless: each request is executed independently

connectionless: a new connection is established for every request

HTTPS is SSL (Secure Socket Layer) protocol, encryption will be done

MIME (Multipurpose Internet Mail Extensions)

type/subtype;parameter=value

text/plain;charset=UTF-8

request header will pass information about request

request body will be absent in case of GET request

<div> tag to group the field names and the input fields of our form together

<span> is similar to <div> element, but <div> is a block-level element whereas <span> is an inline element.

<html>

<head>

<meta>

<title>

<style>

<link>

<script>

</head>

<body>

<form>

<div>

<Label>

<Input>

</div>

<div>

<label>

<select>

<options>

</select>

</div>

</form>

</body>

</html>

select

datalist

only step signifies the interval.

“max”, “pattern”, “value” are designated to “set upper limit”, “matching data value”, “setting default value” respectively.

<a>

href

target <https://www.w3schools.com/tags/tryit.asp?filename=tryhtml_a_target>

\_self(def) -same window

\_blank -new tab

\_parent -parent frame

\_top -same tab

# CSS

Adding CCS to HTML

Inline (<input style = "color:white">)

Embeded (<style>p{color:white}</style>)

External (<link href="cssfile.css" rel="stylesheet">)

If all the three are used Inline conditions are applied

Incase of Embeded and External, the latest declared will be applied

If for same tag same type of different conditions are applied then,

it will take the combo of all of them, while in conflict the latest values.

Type selector < Class selector < ID selector

With pseudo-class selectors we can style the elements based on their position within the Document or based on their interactive state

<a> :link :visited :hover :active MUST be in ordered format to work properly

<input> :focus :enabled :disabled :checked :intermediate :required

CSS Box

MARGIN

BORDER

PADDING

CONTENT

PADDING Top, Right, Bottom, Left (clock wise)

annotations : able to define the data type of the VARIABLE

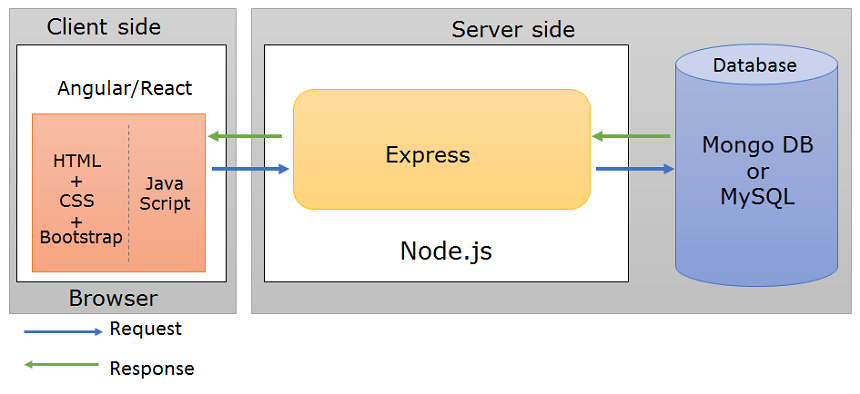
eg: var i : number , var s: string

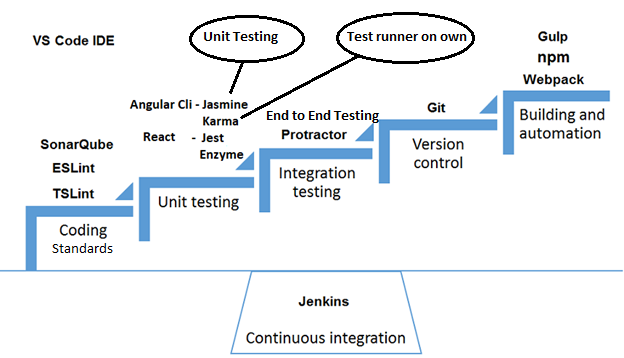
eg:- function f (a: string) : string{}

generics - parameter type dynamic so that function can be reusable with any datatype

eg: function funarray<T>(s:T[ ] ):T[ ]{}

Decorator: All a decorator is, is a function that takes a class as an argument





## CSS Positioning

Default position is static or initial (so whatever we mention top,left etc. it will follow page flow)

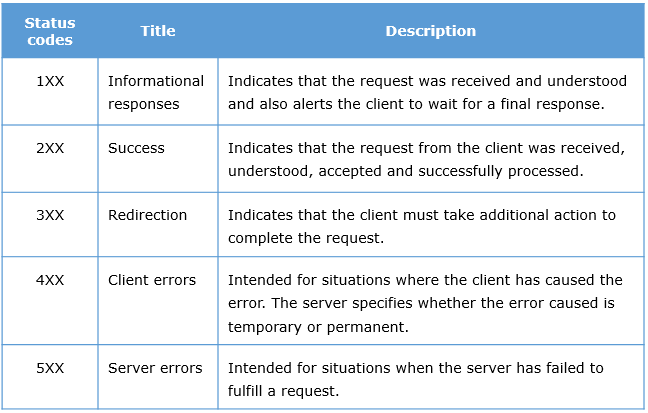
Absolute – similar to static but considers positioning wrt to window(used to keep div anywhere)

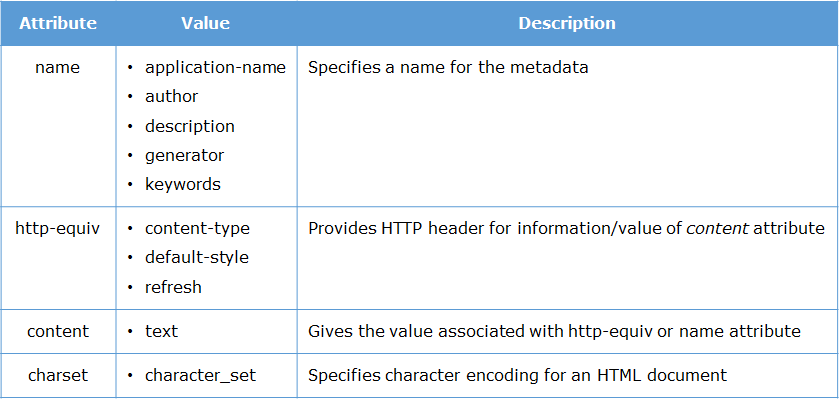
Fixed – if we mention top 0 then it gets fixed there no matter if we scroll (used for navbars)

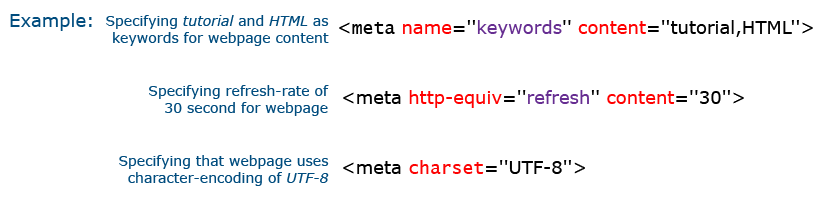
Relative – it considers top, left etc. attributes values relatively from parent position

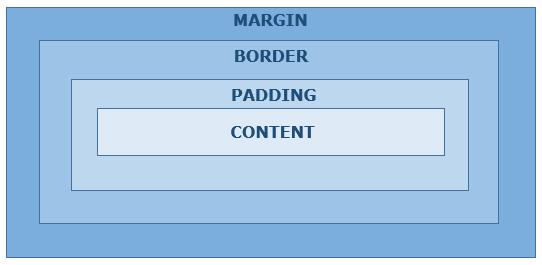
Sticky – Combination of relative and fixed, if we give position:sticky then we need to mention what parameter it should consider to fix till then it acts as relative

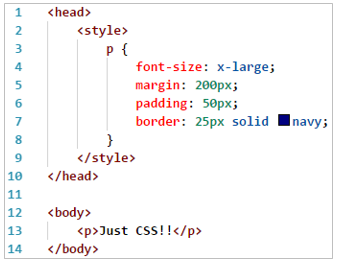
Eg: .navbar{ position:sticky;top:5px} so till navbar top reaches 5px it acts as relative and goes with scroll flow but once top reaches 5px then it will get fixed at that position

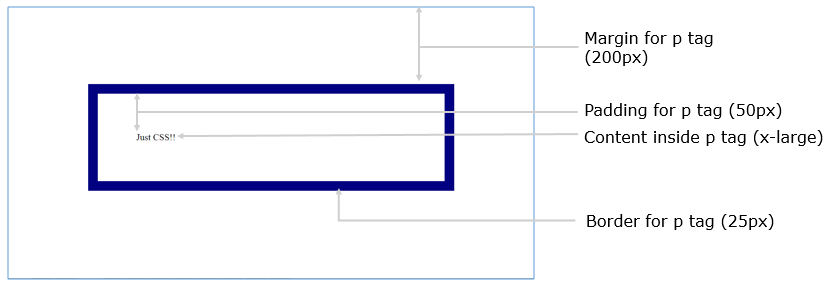












Directives used to change DOM (document object model)

structural directives - \*ngIf, \*ngFor, \*ngSwitch(A Structural directive changes the DOM structure by adding and removing DOM elements.) it won’t hide directly removes from tree. ngFor don’t work on iteration of strings, must be an array.

attribute directives - [ngStyle],[ngClass](Attribute directives changes the appearance or behaviour of a component or element.)

Animations in CSS are done using key-frames.

@keyframes exampleAnimation2 {

0% { background-color: red; }

25% { background-color: yellow; }

50% { background-color: blue; }

100% { background-color: green; }

}

div {

width: 100px;

height: 100px;

background-color: red;

animation: exampleAnimation 4s 1(iteration count) ease-in;

}

Media queries are used to specify how the web page should look on different sizes of screens

@media only screen and (max-width:500px){

body{

background-color: blue;

}

}

Syntactically Awesome Style Sheets(SASS) is considered as a CSS pre-processor and is a stylesheet language .(extension .scss)

used for declaring variables and reusing them.

$default-background-color: white;

.tour {

.tour-card {

border: 1px solid black;

background: $default-background-color;

}

}

used for nesting classes

.content-area {

display: flex;

flex-wrap: wrap;

.main {

display: flex;

flex-direction: column;

flex:8;

.tour {

display: flex;

}

}

}

command to generate css file from sass

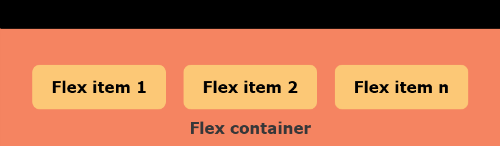
sass scss-demo.scss output.css

## Flex

To use flex first we need a parent flex container( propert display:flex) then its child items become flexible. Flex container contains property like flex-direction:column,column –reverse(elements reverse) or row, it containes flex-wrap:wrap property, if we not mention this then if we increase width beyond flex container size then no change will happen, if we mention wrap and increase size then flex-container will adjust to it without compromising on flex item size (wrap-reverse also there to wrap in reverse order, no wrap also there), flex-flow:row wrap, it is shorthand property to set both flex-direction and flex-wrap at a time. **justify-content**: center or flex-start or flex-end; this property states how items should be justified horizontally , **for vertical we use align-items**:center, flex-start, flex-end, stretch (stretches till container size),baseline(stretches according to baseline), space-between, space-around(for spaces btw items)… All these are flex-container properties.

For flex item properties are order: 2 (means this item is placed in 2 order, can specify the order), flex-grow:4 (by default value is 0, we mention 1 then it doubles (100%+) but it depends on other items too, like all other are 0 and if one item as flex-grow 1 then it will occupy entire space), flex-shrink:2(by default value is 1 so if we give 2 then it will shrink by half, if we give 0 then it will grow double like divide becomes multiple and grows), flex-bases:200px similar to grow but gives px here.

Flex: 0 1 200px it is a shorthand property of flex-grow flex-shrink flex-basis. Flex is very confusing because grow and shrink is based on other item properties also so its better use same flex of every item and then change size accordingly. Align-self : center, flex-start… aligns items within & overrides container align-item properties. Use text-align:center inside div to make content inside item center



Position : **static** (will not get effected by top,left,right,bottom properties)

**Relative** (gets effected if we give left:30px then this will move by 30px)

An element with position: **fixed**; is positioned relative to the view, which means it always stays in the same place even if the page is scrolled (we can use top left.. properties to set its position then fixed)

Postion: **absolute** , this will take top,left,bottom,right properties initial state from its parent. If we give right:0px then it will move right to its parent extent not to window right

Position:**sticky**, this will reacts normally like relative but once window is scrolled down to this extent then it will stick to window like fixed, we have to mention atleast one position to work this like after scrolling stick to top so top:0; safari browsers don’t support this so mention position:-webkit-sticky along with position:sticky, top:0.

When elements are positioned they may overlap, the **z-index** specify the stack order of element (which element should be placed front or behind). If give **z-index : -1** in css then its placed behind. To make image blur and transparent use opacity like opacity:0.3

To make it a center give display: block; margin-left: auto; margin-right: auto;

list-style-type: none; this will remove bullets of list(ul, ol – li)

display : inline (by default all li are block elements here we remove new line formation so comes side by side in navbar mainly) another alternate method is use float: left this will make li into inline and keeps all to left, display:block makes whole thing as a block so we can give margin,padding and makes it clickable. (so by display:block or display:inline-block and float:right or left comes very handy in converting the li elements into horizontal things). Resize:vertical or horizontal or both allows us to resize in UI (like text-area input field happens), overflow:auto makes the scroll bar extend (this need to be given to make resize works).

# **MONGO DB**

Use DatabseName // db.dropDatabase()

db.createCollection(“collection name”) , to delete use db.collectionName.drop()

Mongodb:

db.product\_catalog.insert (  //db.collection name. operations

    {

        prodid:7000010,

        prodname:"nosql distilled",

        publisher:"Addison-Wesley",

        genre: {academic: "technical"},

        ISBN:1234567,

        price:400

    })

Insert acts like both insertone and InsertMany, output WriteResult({"nInserted":1})

db.product\_catalog.find({

$and:[

{manufacturer:"apple"},

{colors: { $all:["black", "silver"] } }

]},

{\_id:0, prodname:1, colors:1, price:1}

).pretty()

$all means should contain all mentioned, $in anyone matches,

$and, $or, $not,

$gte, $ne

db.product\_catalog.updateMany(

{ "price" : { $gt : 80000 }, "manufacturer" : "apple" },

{ $set: { "prodname" : "iphone 7 plus" } },

{ upsert: true }

)

If upsert : true mentioned then if record not found it will create a new record

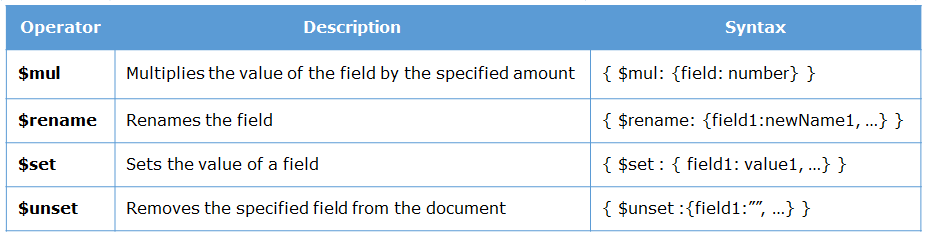
db.product\_catalog.update(

{ ISBN: 18407806 },

{ $inc: { price: 50 } }

)

$inc (+ or -), $mul (multiply or divide), $rename (change field name) , $set (sets field value), $unset (removes specified field from document)

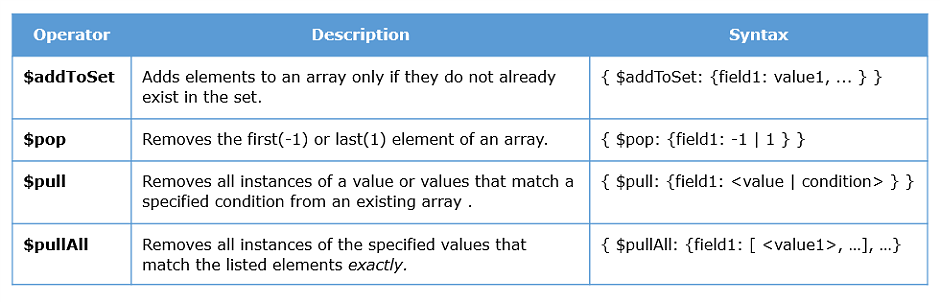


db.product\_catalog.update(

{ prodid: 7000001},

{ $push: { colors: "white" } } )

To push value into an array in db (if doesn’t exist create a new one)



db.stores.update(

{ }, #if we left search conditions as empty, this will match with all records

{ $pull: { fruits: { $in: [ "apples", "oranges" ] }, vegetables: "carrots" } },

{ multi: true } ) //after the update it will remove apples, oranges in fruits and carrots in veggies list.

We use $elemMatch to match the conditions DB: { \_id: 1, results: [{score:8,item:”A”}, {score:5,item:”B”}]}, { \_id: 2, results: [ [{score:9,item:”C”}, {score:8,item:”B”} ] }. db.scores.find({ results: { $elemMatch: { score:{$gte: 8} , item: "B" } } } matches \_id:2 record

Delete:

db.product\_catalog.deleteMany(

{ price: { $lt : 1000 } } )

Also has deleteOne // these are old versions, new version has db.name.remove(<query>,{justOne:true}) , if justOne true is not given then it acts like deleteMany.

To delete or perform any specifc n no of times (use .limit(number)), we either can run a for loop

db.collectionName.find().limit(N).forEach(doc => //first we are finding only N number of records

{ and then forEach record deleting below with its \_id

db.collectionName.remove({\_id:doc.\_id}) this forEach is js code not mongo so valid in js files

}

) or another method

removeIdsArray = db.collectionName.find({}, {\_id : 1})

.limit(100)

.sort({timestamp:-1}) //this makes the last 100 records from time

.toArray()

.map(function(doc) { return doc.\_id; }); // Pull out just the \_ids

Then pass the returned ids as an array in below query

db.collectionName.remove({\_id: {$in: removeIdsArray}})

$position:0 this means do operation on the 0 index of array, if db is {“\_id”: 1, "scores" : [ 100 ] }

db.students.update( { \_id: 1 },{$push: { scores: {$each: [ 50, 60 ], $position: 0 }}})//[50,60,100]

$sort:1 will give output in ascending order inside array, $each is do operation like push each object

Aggregation :

db.product\_catalog.count( { "categories.sub": "smartphones" })

db.product\_catalog.distinct( "manufacturer" )

Example:

db.product\_catalog.aggregate( [

{$project: { \_id: 0, manufacturer: 1, price: 1 } },

{$group: { \_id: "$manufacturer", totalPrice: { $sum: "$price" } } } //($ is used to represent)

] )

Example: (similar to find query)

db.product\_catalog.aggregate(

{ $match: {$and: [{ manufacturer: "lenovo" }, {price : {$lt:10000} } ] } } )

Example:

db.product\_catalog.aggregate( [

{ $sort : { price: 1 } },

{ $limit: 5},

{ $out: "FiveCheapestMobiles"}

] )

Example:

db.product\_catalog.createIndex(

{ price:-1 })

Example:

db.product\_catalog.createIndex(

{ price:1, rating:-1 })

Example:

db.product\_catalog.createIndex( {categories:"text"}) //collation is used to specify language specific rules for string comparison in creating indexes, if we don’t specify any then takes default, db.product.createIndex({categories:”text”},{“collation”:{locale:”fr”,strength:1}})//verify website

db.product\_catalog.find({ $text:{ $search:"smartphones"}})

Example:

db.product\_catalog.getIndexes()

Syntax:

If you know the name of the index

db.collection\_name.dropIndex( "index\_name" )

OR

For ascending indexes:

db.collection\_name.drop\_Index( { field\_name : 1 } )

OR

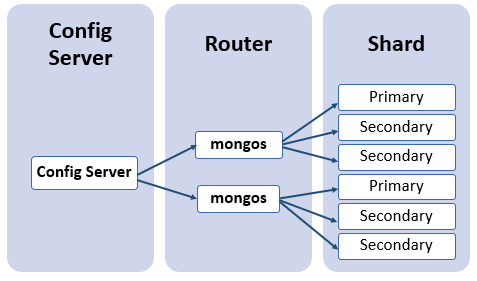
For descending indexes:

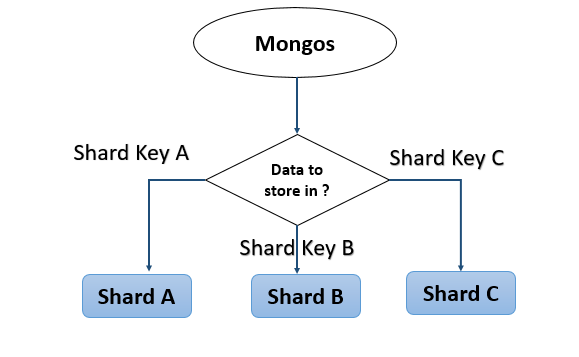
db.collection\_name.drop\_Index( { field\_name : -1 } )

Syntax: to delete all indexes

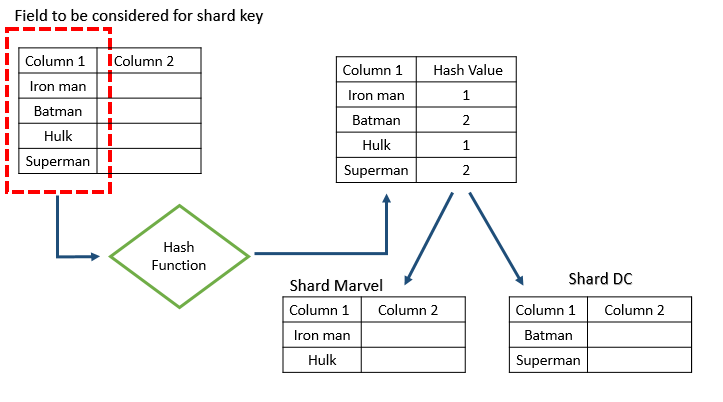
db.collection\_name.dropIndexes()

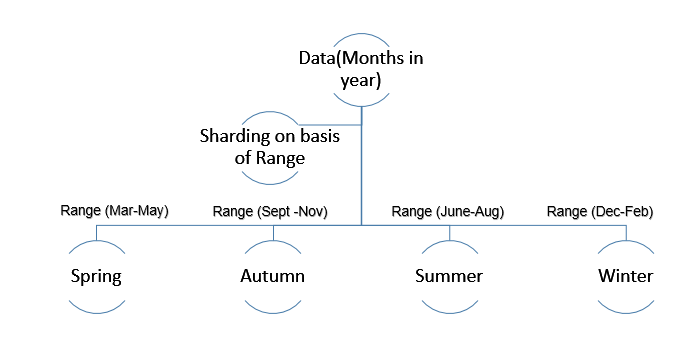
normal numbers we give are 32bit (NumberInt), if we want to insert long 64 bit numbers then have to mention NumberLong(22121212121). Same for 64 bit float NumberDecimal(“110.223”), For eg: there are 2 records of price value 10 and NumberLong(10) then if we make query find({price:10}) then it will give only 1 record as result it wont consider NumberInt 10 not equal to NumberLong(10), same for NumberDecimal and normal float number. If we make eg: $inc operation they turns normal





MongoDB Compass can be used as GUI for mongo db collections and data. MongoDB Atlas is a cloud based mongo service provider.





English - is/am/are/was/were + used to + ing verb

superlative degree follows in (if it is place/group) else it follows of

eg: the shortest boy of/in the school (ans: in)

if ends with ce then it is abstract noun

present perfect/present perfect continuous we will use since not for

# **JAVASCRIPT**

* JavaScript is an interpreted language
* JS has 5 data types: number, string, undefined, null, object

# **Functions**

* Functions are objects in JS
* Functions can be stored in variables, passed as parameters or returned as values
* Functions without names are called anonymous functions
* Arrow functions are shorter way to write anonymous functions
* Variables can either be in global, local or block scope

Var x = function () {} // x is being assigned to anonymous function, we can send these as parameters

Function x(){}

X = ()=>{} //arrow function , x = \_=>console.log(\_) //function for single parameter

If function takes another function as parameter then we call it as higher order functions, so we can pass that parameter function as anonymous function.

Arrow functions are mainly used to bind **this** object in classes (see it in typescript below)

1. Functions are actually Objects. That means a function can be stored in a Variable.
2. We can also pass Functions as a parameter to another function.
3. Before any function executes, all the Local Variables are **Hoisted**in the function. Hoisting is a phenomenon, where no matter where the variable is declared inside the function, they are all ***pushed as the first statements inside the function*** during the function execution.
4. However, only variable name is hoisted and ***not its value.***
5. The variable ***i*** has been declared with ***var*** keyword, it is ***accessible throughout the function***. Var is lexical scope (accessible outside block). Let,const are block scopes
6. A ***variable with a block scope (let, const)*** is accessible only within the block of statements and not throughout the function.
7. ***const*** is a keyword which is also used to create a block scoped variable. But the difference between const and let is that, a const variable ***cannot be modified***. It is Constant. (also verify at the datatypes table at the end after DOM in angular)

Objects have ***properties and methods***. JavaScript provides many standard **built-in** objects. In addition to that it also provides an option to create **user defined** objects.

Some of the commonly used built-in objects are:

* Array
* Date
* String

# **Array**

* .forEach()
* .map()
* .filter()
* .find()
* .push() //added to last
* .pop() //removes last
* .splice(startingIndex, NoOfElementstobeRemoved, elementsWantToReplace(optional))
* .slice (startingIndex including, ending index not included)
* .join() //pushes at index 0

To insert at specified index: ourArray.splice(insertAtIndex, 0 , insertedValue).

1. Consider the **forEach()** function of an array. This function takes another function as parameter and invokes the function for every item in the array. Doesn’t return anything

placesToVisit= ["Paris", "New York", "Switzerland"];

placesToVisit.forEach(place => console.log ("Trip to " + place));

// Trip to Paris

// Trip to New York

// Trip to Switzerland

1. We know that an array object has a **.map()** function that creates a new array based on what the passed callback function returns value.

placesToVisit= ["Paris", "New York", "Switzerland"];

placesUpperCase = placesToVisit.map(place => place.toUpperCase()); //return is not mentioned here cause in arrow fns if it is single line without {} it will automatically returns else we have to mention return for sub array creating functions. If we don’t return it will add undefined here.

console.log(placesUpperCase);

// ["PARIS", "NEW YORK", "SWITZERLAND"]

1. We know that an array object has a **.filter()** function that returns a filtered sub array based on what the passed callback function return true, other than true null, false it won’t add.

placesToVisit = ["Paris", "New York", "Switzerland"];

filteredPlace = placesToVisit.filter(place => place.length > 5);

console.log(filteredPlace);

// [ 'New York', 'Switzerland' ]

So the .map() will assigns the returned value, .filter() will assign if callback returns true to new array.

**.find()** function that returns the first element in the array based on which record is return true by callback function. Similar to filter but it stops at first element. So no array is created.

placesToVisit = ["Paris", "New York", "Switzerland"];

findPlace = placesToVisit.find(place => place.length > 5);

console.log(findPlace);

// "New York"

We can chain these builtInfunctions like

placesToVisit.filter(place =>place.length>5)

.map(place=>place.toUpperCase())

.forEach(place=>console.log(place))

**Palindrome**

function isPalindrome(s)

{

var reversedText = s.toLowerCase().split('').reverse().join('');

return s=== reversedText;

}

console.log(isPalindrome("level"));

# Objects

Creating **Object** using object literal

var empOne = {

name : "John",

empNumber : 1001,

emailId : "John@gmail.com",

swipeIn(){console.log("Swipe In by "+this.name)}

};

**Accessing object properties**

object.property

object[property]

**Iterating an object**

for..in // to iterte the object properties

object.values(ObjectName) //gives all the values in the object as array

Using Object Destructuring we can destructure an existing object into variables. If the var is prefixed by 3 dots, then it is a Rest Variable and can store more than one property. It has to be array only

{ a, ... rest } = empOne

console.log(a)

// 'John'

console.log(rest)

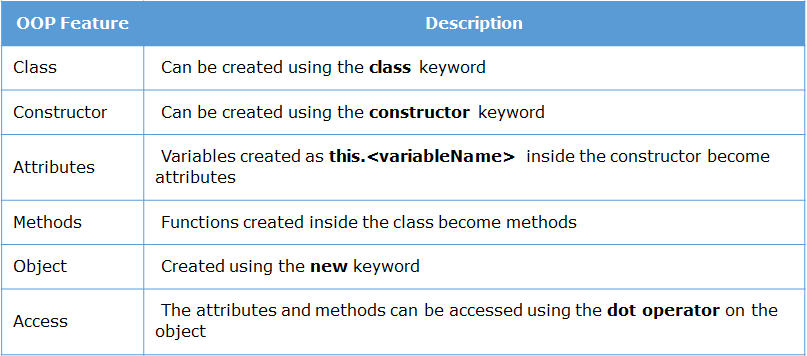
/\*{

empNumber : 1001,

emailId : "John@gmail.com"

}\*/

# Class



## this

console.log(this) //gives global window object Window {… }, if we print inside function still same result. This states all are in global window object. But if we create a new class and prints this then

class Demo { getThis(){console.log(this)} }; var x = new Demo(); x.getThis() // output Demo { }

This states now we are inside Demo object, Till now everything is good but if we print this inside a function within another function(BuiltInFucntions forEach,map etc are also another functions) of Demo class we will get Window { } object again. To overcome this unusual behaviour we use arrow functions, it will bind the this object inside the functions

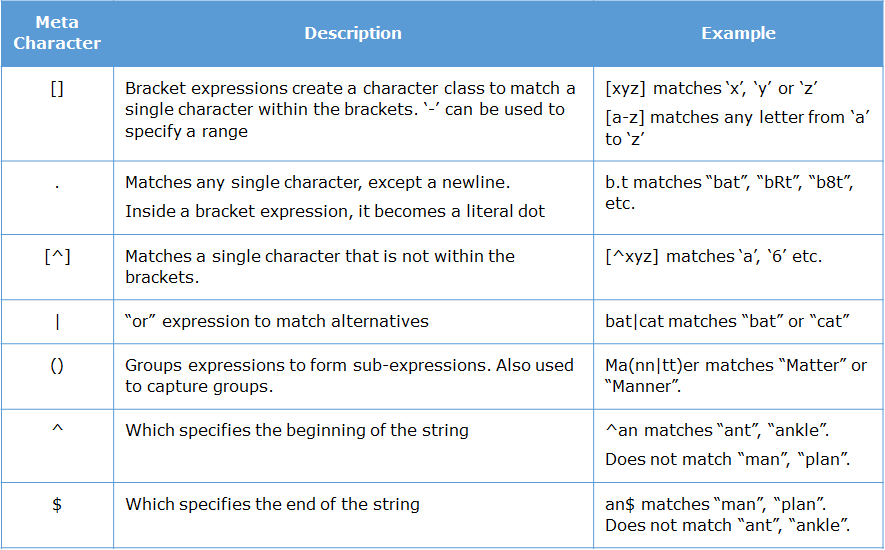
class Demo { getThis(){get2This(){console.log(this)};get2This() }; var x = new Demo(); x.getThis() //Here the getThis will invoke get2This which will print Window object

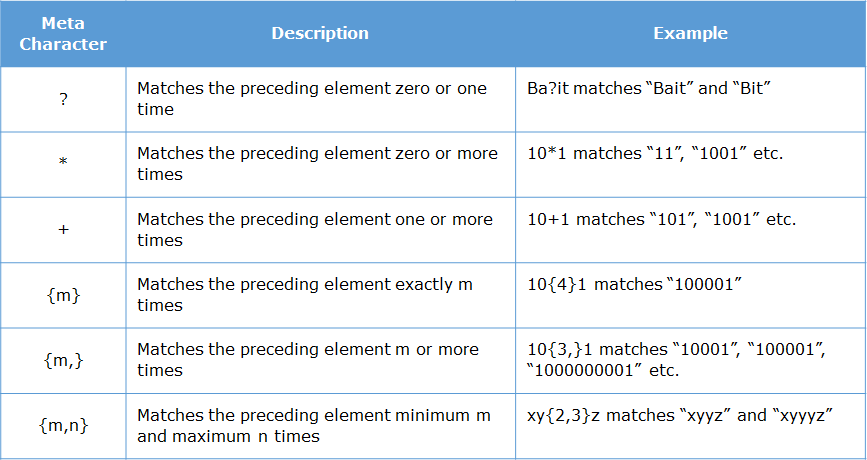
class Demo { getThis(){var self = this; get2This(){console.log(self)};get2This() }; var x = new Demo(); x.getThis(); //since we captured this before invoking 2 function we will get Demo { }. (not ideal way)

class Demo { getThis(){get2This()=>console.log(this);get2This() }; var x = new Demo(); x.getThis()

// We will get output Demo { } cause arrow function binds this object,getThis arrow fn is optional

# RegEX





\w to match [A-Za-z0-9\_], \W is negation of \w. \d matches [0-9] and \D except [0-9]. \b is used like str=”HELLO LOOK”; str.search(/\bLO/); this searches LO at beginning of word which is index 7, str.search(/LO\b/) it searches LO at ending of string which is 3. \B is opposite /LO\b/ gives LO matching position but not at ending so 7. \0 matches null, \n new line in str. In regex special symbols like . + ? etc. have special meanings but if we want them as normal matching then use \. \+ \? Etc.

# URI and URL difference

[example.com/Project/posts](http://www.example.com/Project/posts) – URI (uniform resource identifier) can identify but not locate

http://www. [example.com/Project/posts](http://www.example.com/Project/posts) – URL ( UR Locator) can be used to locate and identify a resource. So there can be many URI’s but only few URL’s just like many persons with same name but identical person with exact location, name, address etc. only one.

# WEB API

In fact console.log() is actually not part of the language. It is provided by the browser as part of its API. It is through that API we can interact with the browser and print something on the browser's console.

There are many Web API. Some of which we will cover in the course are:

* XMLHttpRequest
* DOM
* Notification
* Storage

**Note:** Since these API's are provided by the browser, they will not work in Node. Except console.log works in Node because node.js is based on chromes v8 engine which is used from browsers

XMLHttpRequest and DOM are Asynchronus Javascript And XML (AJAX) in nature

setTimeOut(fn(),timeInMilliSec) is DOM api following AJAX technique

Same, XMLHttpRequest and its methods like http.open(‘GET’,url) follow Asynchronus i.e AJAX techniques

To make sure the function executes once the response is received we use callbacks, if there are too many callbacks nested up like callback has another callback (callback hell) to avoid this problem we use promise.

function getTrip(){

return new Promise(function(resolve){

setTimeout(function() {

resolve("Lets go to Trip");

}, 2000);

});

};

This code promises a states that if everything goes well then it will resolve the “Lets go..” data

function bookFlight() {

return new Promise(function (resolve) {

setTimeout(resolve(5600), 2000);

})

}

function bookHotel(flightPrice) {

return new Promise(function (resolve) {

setTimeout(resolve(7000 + flightPrice), 1000);

})

}

function getTotal(){

bookFlight()

.then(function (flightData) { return bookHotel(flightData) })

.then(function (cumulativeData) { console.log(" Total is " + cumulativeData) })

}

getTotal()

we can use await inside async function to make code wait till promise gets resolved, above code can be written in

async function getTotal(){

var flightData=await bookFlight();

var cumulativeData=await bookHotel(flightData);

console.log(" Total is " + cumulativeData)

}

Suppose there is a JSON file in ../json url then we can access through XMLHttpRequest

Function getData(url){

var xhr = new XMLHttpRequest();

xhr.open(‘GET’,url);

xhr.onload = function () { // Invokes after getting response

var json = JSON.parse(xhr.responseText)

document.getElementById(“response”).InnerText = JSON.stringify(json);

}

}

# UNIT TESTING:

. JS doesn’t have any inbuild compiler to detect errors, we can only find them directly in the browser. So, unit testing is very important during coding in the Js. Keeping console.log() statements in the code in every function is lengthy process so automated testing JASMINE is used. Test suits created are executed with the help of karma.

Eg for test suite:

describe('TotalTravelFare calculation Suite:',function(){

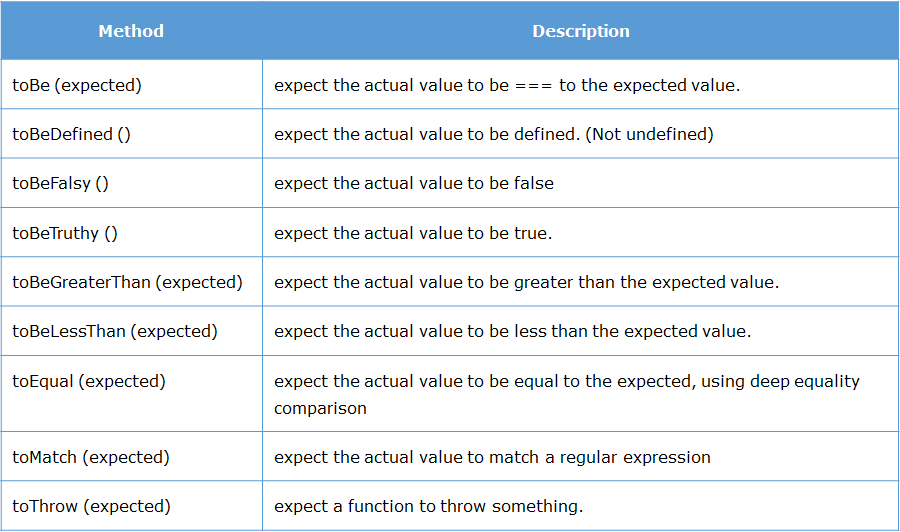
it('Test Case 1: Inputs are correct',function(){

expect(totalTravelFare(1000,20)).toEqual(1200);

});

})

describe,it,expect is global jasmine function, toEqual are Jasmine matchers. Others are



Can use **not** like – expect(“sd”,function(){expect(12).not.toBeLessThan(10)})

To maintain resources and not to invoke or access common functions or data we use beforeEach and afterEach

beforeEach(function(done) {

setTimeout(function() {

string = 'Stormhold';

}, 1500);

});

describe('Value of n', function() {

var n = 0;

afterEach(function(done) {

setTimeout(function() {

n++;

done();

}, 1500);

});

it('is 0', function() {

expect(n).toEqual(0);

});

it('is 1', function() {

expect(n).toEqual(1);

});

});

After the test cases are written in .js file to run those we need KARMA (Test runner tool)

npm install -g karma jasmine-core

Karma is:

* A tool that spawns a web server which executes the source code against the test code for each browser connected.
* When executed, it automatically captures the browser specified by the developer during Karma configuration.
* It then displays the results on the command line.
* It watches all the files specified within configuration file and if there are any changes, it will trigger the corresponding spec again on the browser.

Conclusion: We create test cases using jasmine global functions and run them on KARMA tool by command ( karma init ) to get karma.config file with all dependencies followed by (karma start).

# Document Object Model

DOM API – Using the DOM API we can interact with the HTML elements and the browser window using JavaScript. DOM stands for Document Object Model. It models the HTML document into an object. That means, everything we see in the HTML page is represented as objects in JavaScript.

**DOM**is not a programming language; it is an interface through which JavaScript access the elements of HTML pages.

One of the most common methods associated with the document object is:

**getElementById(id):** It is used to access element by its id.

**getAttr("attributeName"):**It is used to access the tag attribute of the DOM object

**innerHTML:** innerHTML is used to set or get the HTML content of the element.

**innerText:** innerText is used to set or get the Text content to element.

**value:** value is used to get or set value of the element.

**alert("message")** - this will display an alert box with the given message

**confirm("message")** - this will display an confirm box with the given message

**write("message")**- this will overwrite the existing HTML content of the page with the message given

console.log(document.getElementById("username").getAttribute("placeholder"));

console.log(document.getElementById("password").value)

//All elements are also stored as arrays. We can also access based on the index position.

console.log(document.forms[0].elements[1].getAttribute("maxlength"))

directly like document.formName.fieldName.value

<form><input type=”text” name = “iname” id=”ID”></form>

Document.form.iname.value

Also we can get by: const field = document.querySelector(‘.className or id’) (it runs a query and gives the first matching css class or id with string passed) to return all use querySelectorAll

Field.addeventListener(‘click’, () => { }} (will invoke if field is clicked)

Field.classList.add(‘new class’)v(adds new class to the field)

We can get details of the element on which the event took place using **event.target**

<input type="radio" name="gender" value="male" onclick="display(event)">Male

<script>

function display(e){

console.log(e.target.name);

console.log(e.target.value); }

</script>

Suppose there is Div1 which has onclick event and has child Div2 inside it which also have onclick event then if we click on div2 then both div2 click event followed by div1 click event gets invoked this is called **Event Bubbling**. If you want to prevent this we can use **event.preventDefault()** .

Tip: Just to make page reactive use container/fluid class at begging and then create div with class row and inside it use classes col-md-4 etc then create forms etc in it (bootstrap)

# Session Storage

Session storage is a Web Storage API introduced in HTML5

We can store the data using web storage in three ways:

1. For particular session i.e. **session**storage: For session storage, *sessionStorage*object is used (single tab, few kb storage, ends if browser closed or session cleared)
2. Across sessions i.e. **local**storage: For local storage, *localStorage*object is used (upto 8MB stores in client storage permanently until cleared)
3. Cookies are attached to each request we send, used by both frontend and backend, small size else it slows down request speed. We can access by **document.cookie**

Both these objects are created by JS runtime engine of browser.

Web storage API provides following four methods for managing data:



# Notification

We can create desktop notifications in JavaScript using the Notification API

To gain access from user: Notification.requestPermission()

To display notification:

1. function notify(){
2. Notification.requestPermission();
3. if(Notification.permission === "default"){
4. alert("Please grant permission");
5. }
6. else {
7. var notify = new Notification("New Mail",{body:"You have 1 unread email"});
8. }
9. }
10. Possible values for permission are denied,granted,default, states current choice

6. body – Defines notification body

# Cross-site scripting (XSS)

Cross-site scripting (XSS) is a security bug that can affect websites through injecting code

Eg: <img src = 1 onerror=”s=document.createElement(‘script’);s.src=’../evil.js;document.body.appendChild(s)’”>

<img src = x onerror = “alert(document.cookie)”> //gives all values in cookie

This can be prevented by running regEx on every input fields we provide before processing that data(validating input), using a template with context-aware by auto escaping or manually doing the escaping on every input data (<script>alert(‘hello’)</script> is converted into &lt;script&gt;alert(‘hello’) &lt;/script&gt; this is called escaping which is encoding actually)

Objects:

Objects are mutable but you cannot create a replica of it

Eg: var emp = {name : “Pavan”}

Var x = emp; // this is calling emp by x, not creating a new x

x.name = “XYZ” // this will change both the emp.name and x.name value

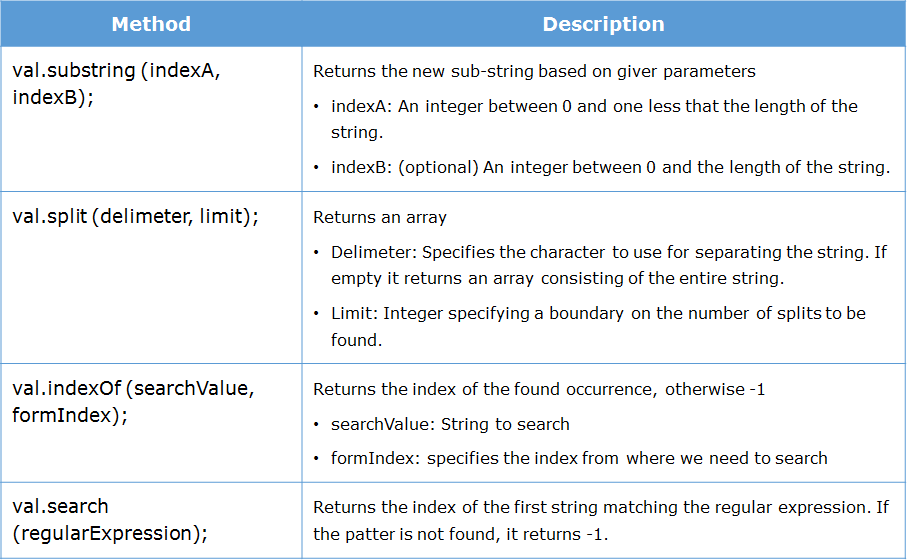
If we want to change the key name just assign old key value to new key value and delete the old one

Eg: emp.fullname = emp.name

Delete emp.name

To know whether object emp has ‘example’ property then emp.hasOwnProperty(‘example’) returns true or false.

# String operations



Val= “012345” 🡪 val.substring(1) = “12345”, val.substring(1,4) = “123”

# WEB Worker

When we need to do I/O operations or run scripts in the background simultaneously then we use web workers.

Eg: this is worker.js file which sends message through **postMessage** method of var i continuously with 500millisec gap

var i = 0;

function countNumbers(){

if(i < 100000){

i = i + 1;

postMessage(i);

}

setTimeout("countNumbers()", 500); // Waiting for 500millisec before running this script again

}

countNumbers();

HTML page to use this

<html>

<script type="text/javascript">

if(window.Worker){

var worker;

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

worker.onmessage = function(counter){

document.getElementById("countervalue").innerText = counter.data;

};

} else{

alert("Web worker is not supported in your browser");

}

</script>

</head>

<body>

<div id="countervalue">

<!-- Messages received from web worker will be displayed here -->

</div>

</body>

</html>

# Closures

If a function takes another function as parameter then it is called Higher order functions. If functions have another functions inside it then it is called closures.

A closure is an inner function that has access to the variables present in the **lexical scope**(parent scope) of the outer/containing function. Therefore it can outlive the lifetime of the outer function and hence can continue to access the variables of the outer function.

function outer() {

var count = 0;

function inner() {

count=count + 1;

return count;

}

return inner;

}

private\_function=outer();

console.log(private\_function()); // 1

console.log(private\_function()); // 2

console.log(private\_function()); // 3

Here we can access inner function without invoking outer thus it can outlive outer fn.

# **Immediately Invoked Function Expression (IIFE)**

If function (mostly anonymous) is invoked immediately after it is created then it is call IIFE

Eg: (function (num1, num2){return num1 + num2;})(100, 200) // 300

So if we invoke () immediately after the anonymous fn then it will run and gives result.

Another example: Note: function has to be inside () and next function has to be direct

var outer = (function () {

var counter = 0;

return function () { return counter += 1; }

})(); // var outer = (...an anonymous function...)();

console.log(outer()); //1

console.log(outer()); //2

console.log(outer()); //3

here also anonymous fn gets invoked by () in the end and it returned a function which gets called in the console.log by outer(). It’s an example for both IIFE and closure too.

Note: IIFE means ()() it should not be ();()

# Object with Function

An object can be created by using **constructor**function. The syntax is:

function Name(param1,param2,param3) {

this.param1 = param1;

this.param2 = param2;

this.param3 = param3;

this.swipeIn=function(){console.log("Swipe in by "+this.param1);}

};

var obj = new Name(p1,p2,p3);

# ParseInt and ParseFloat

Used to convert string to intiger

parseInt("5"); // 5 parseFloat("1.23"); // 1.23

parseInt("5.5"); // 5 parseFloat(".123"); // 0.123

parseInt("5p0"); // 5 parseFloat("1.23abc"); // 1.23

parseInt("p50"); // NaN (Not a Number) parseFloat("a123") // NaN

tip: typeof(NaN) = number //NaN = 0, this 0 actually like false 0

typeof(3.14) = number //there is no type called float

# NODE and Express

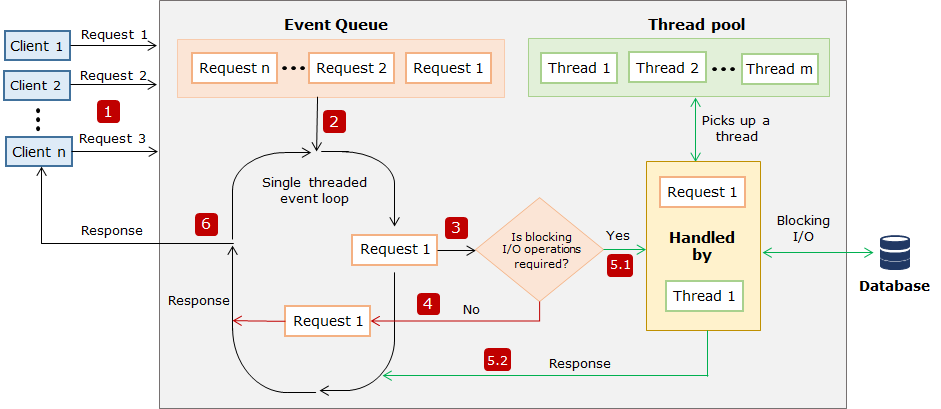
**Node**.**js**® is an open-source JavaScript runtime built on Chrome's V8 JavaScript engine.

Express is the most popular framework for creating web applications in Node.js. It is lightweight and provides easy connectivity with database like MySQL and MongoDB. Using Express we can handle requests, create views and manage routes. It contains main application.js to start server etc, routes.js for routing, Business Layer (BL) files, Database Access Layer (DAL), services etc.

**Reduced resources**, **Library support (NPM), Asynchronous and Event-driven**, **Wide client side (Angular, react) and database connectivity (Mongo, MYSQL)**

* V8 is Google's open source high-performance JavaScript engine, written in C++ and used in Google Chrome, designed to improve the performance of the JavaScript execution in browsers.
* At the time of execution, it converts the JavaScript code into machine code using **Just-In-Time (JIT) Compiler**, to achieve high speed. Therefore byte code or intermediate code will not be created

Node.js environment is created based on **Single Threaded with Event Loop Model** which is built using JavaScript's callback mechanism.



First we need to express generator to create project, (npm i express-generator –g followed by express appName )this creates skeleton including package.json.( this is another way :To begin with we create package.json file with npm init command with all the basic info(version,author, dependencies etc) ). then we install required like npm i directly will install mentioned packages in package.json dependencies or npm i express body-parser cors.

# Modularization

Modularization is a software design technique in which the functionality of a program is separated into independent modules, such that each module contains a desired functionality.

Syntax to export: exports.<module\_name> = <module\_implementation>

export.addFn = (x,y){return x+y} //just like in js export class abc or export function xyz() or export default fn e()

const myCalculator = require('./calculator'); //just like in js import e,{abc,xyz} from ‘filepath’

myCalculator.addFn(1,2) //3

if there is one export then we can use module.exports = add so if we import by require in another file it will directly refer to add i.e myCalculator(1,2) // 3

# Modules

These modules are used first by assigning them to local variable by require() method

Eg: const express = require(‘express’);app = express(); const fs = require(‘fs’)

Nodemon – it is a package which will automatically refresh the server if any changes occur (just like how ng serve and autosave works in angular), we install it just like express and other modules(npm i nodemon -g) and run the files by command npx nodemon app.js in cli instead of node app.js

fs – this module is used to read,write,append a log file (error , request logger)

fs.writeFile(file\_path, data, callback) , appendFile() (write will overwite the content and creates new if not exist) (callback function is invoked even though the write is success or not)

eg: const fs = require('fs');

let fileWrite = () => {

let str = "Hey, there.. ";

fs.writeFile('./log.txt', str + " At: " + new Date().getFullYear(), (err) => {

if (!err)

console.log('Data Written!');

else

throw err;

})

}

fileWrite();

eg: const fs = require('fs');

fs.readFile('demo.txt', 'utf8', (err, content) => { // charset is optional

if(!err)

console.log(content);

else

throw err; });

## http: (webserver module)

permits us to send data over the HTTP protocol, is used to build a HTTP server

let server = http.createServer((req, res) => {

res.write('Hello World! I have created my first server!');

res.end();

})

server.listen(3000);

console.log("Server started... Running on localhost:3000")

this is creating server in node.js, later this entire thing is replaced by express with simple app.listen(3000) (app.listen takes a callback just like fs.readfile above which is optional)



if we send res.send(“<i>italic</i>”) then it will take it as plain text not as html so to solve this we use response.writeHead(200,{"Content-Type":"text/html"}); now it will print as *italic* , for express it is res.set('Content-Type', 'text/html');

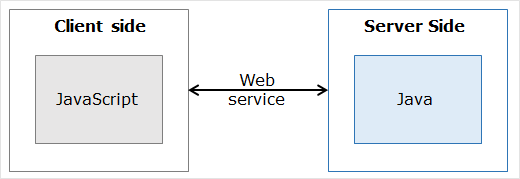
Tip: http and fs are built in modules(provided internally by node.js) directly used by require() but a NPM(provided externally by node.js) module like express has to install and use it

npm install <package\_name>[@<version>] //version optional if not given takes latest

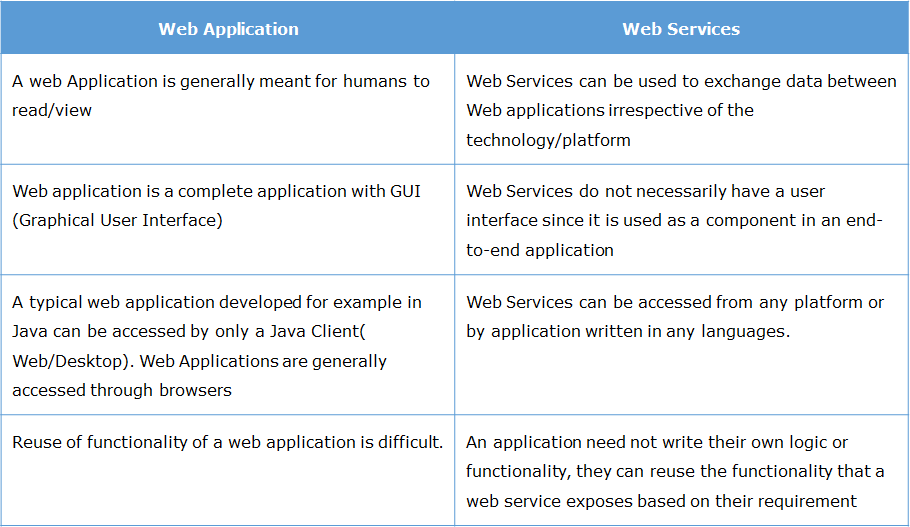
This will create a folder **node\_modules** in the current directory and put all the packages related files inside it.

# Express

Express is a **layer** built on Node.js which helps us to **manage our web server**and **routes**.



If any two different platforms want to communicate or transfer data it can happen through webservice. So http using www is a webservice, google maps in other apps is achievable through webservice.



# RESTful API

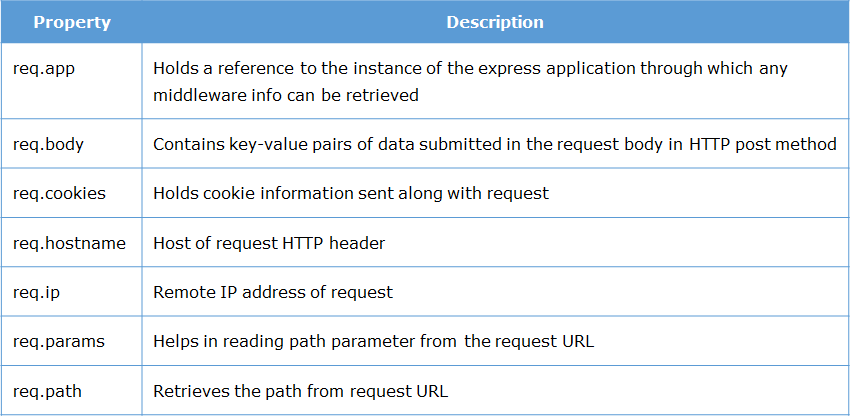
Web services can be classified into two types based on the transfer of data.

* SOAP based web service (XML based)
* RESTful web service (REST  stands for  "***Representational State Transfer***". It is an architectural style for the web which uses HTTP Protocol for communication.)

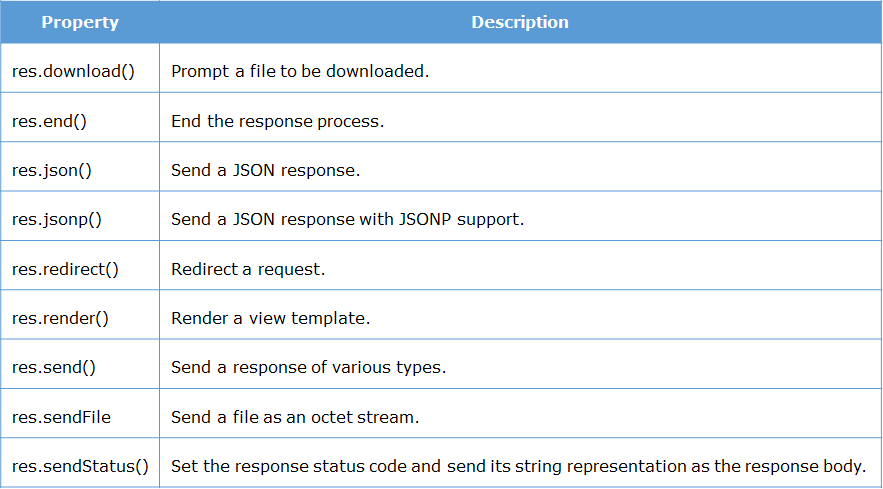
RESTful API is typically used to refer to web services implementing REST architecture.

In RESTful, resources are manipulated using GET, PUT, POST and DELETE methods of HTTP to perform read, update, create and delete operations on the resource.

Objects which can be accessible with request from client



Response objects



Also res.type()

To send a file through response , eg for res.sendFile()

Eg: const path = require(‘path’)

app.get('/register', (req, res) => {

console.log('request for register.html file');

res.sendFile(path.join(\_\_dirname + '/views/register.html')); });

// to join the paths we used path, cause generally outer paths in server changes. \_\_dirname is a predefined variable that provides the absolute path of the resource in which it is used.

Query strings are the data appended as part of request URL. Query strings begin with a question mark and the name-value pair that are separated by an '&'(ampersand).

'***http://localhost:3000/login?username=john&email=john%40infy.com&login=Login***', (%40 is encoded @) encodedURI(normal URI) gives the encoded URI

app.get('/login',(req,res) => {

res.send(`<h1>Hello, ${req.query.username}</br>Your email - ${req.query.email} is registered</h1>`);

});

Syntax: request.query.<querystring name-value>

# Routing

if we use app.get for so many routes it will be messy so we use router class in a separate router.js file with all routes at one place

const router = express.Router();

router.get('/', (req, res) => { res.send('home page') });

we call this router in main app.js file as a microservice i.e app.use(‘/’,router)

main advantage is we can use router method for different alike routes

const express = require('express');

const userRouter = require("./routes/user\_router.js");

const adminRouter = require("./routes/admin\_router.js");

const app = express();

app.use('/user', userRouter);

app.use('/admin', adminRouter); //code is clean and maintainable this way

next() this is used for invoking next matching handler

app.get('/login',(req,res,next) => {console.log('Request to display login page has received');

next(); }); //invokes next matching middleware and response.send

app.get('/login',(request,response) => { response.send('Login Form'); }); (no need to be req,res)

**Array Syntax for chaining of routes Handlers**

let handler1 = (req, res, next) => { console.log('Request to display login page has received');

next();}

let handler2 = (request, response) => { response.send('Login Form');}

app.get('/login', [handler1, handler2]);

or we can directly write route handlers back to back separated by comma

app.get('/login', (request, response,next) => { console.log('Request to display login page has received');

next(); } ,

(request, response) => { response.end(); }); //both handles passed as parameters.

We can group multiple HTTP methods and their handlers, for the same route path using route(), this method can also be used with both app.route() or router.route()

router.route('/login')

.get((req, res) => { res.send('display login form'); })

.post((req, res) => { res.send('process login form'); })

app.get() etc takes numerous functions as parameters like app.get(‘/login’,requestLogger,(req,res)=>{},errorLogger). These functions will be invoked by next()

# Middleware

By middleware we can run set of code in another place by app.use() method. next() method is used to invoke or pass the handler to the next middleware. If we want to specify the method then app.get or app.post, rest all same

app.use(PATH,CALLBACK)… Path is optional here.

## User-defined middleware

Both application level middlewares, routing middlewares are user defined. Chaining of middlewares is done through next() method

let mylogger = (req,res,next) => {

console.log('Request method is '+ req.method);

console.log('Request url is '+ req.url);

next(); }

app.use(mylogger);

app.get('/adminLogin',(req,res) => {

res.send("Admin login Success"); });

app.get('/userLogin',(req,res) => {

res.send("User login Success"); });

app.get(‘\*’,(req,res)=>{ // for other than adminlogin and userlogin

err = new Error(‘unknown link’);

err.status = 404;

next(err); // this will invoke the next match err handler (u can also throw err which will

}) be handled by next error handler) err name can be anything no need to match

app.use((err,req,res,next)=>{ res.send(‘Error: ’ + err.status + ‘occured’)}) // this will get called if error exists, it’s a error handler so placed in the end. Only gets called if err value exists.

Note: error can be captured only if first parameter is error (can be any name like err, er or e etc)

Eg: app.use((myerror,req,res,next)=>{ res.send(‘Error: ’ + myerror.status + ‘occured’)}) //works

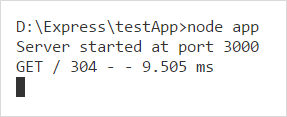
In the same way during throwing error or next(error) the var error name can be anything, no need to match with the error handler parameter, it is only captured by first parameter by error handler.\* If no error is invoked then this handler method is not invoked, so if we keep any simple console.log() this won’t execute if there is no error invoked \*

## Third Party middleware

Morgan – can be used to log requests

Eg: const morgan = require(‘morgan’);

app.use(morgan('tiny'));



Express-error-handler – can be used to log error stacks

const errorhandler = require('express-error-handler');

we can raise error by app.use(errorhandler.httpError(500)) or by err.status = 500 ;next(err) in code where ever required and then we can catch it all errors at once like

handler = errorhandler({handlers:{‘404’: () ={console.log(‘not found’)}},{‘500’ : ()=>console.log(‘internal server error’)}})

app.use(handler);

# BodyParser and CORS

body-parser is a Node.js body parsing middleware which parse incoming request bodies and makes the data readable under the **request.body** property. Without this middleware, data sent through POST request is **inaccessible**. We use it as middleware. i.e app.use()

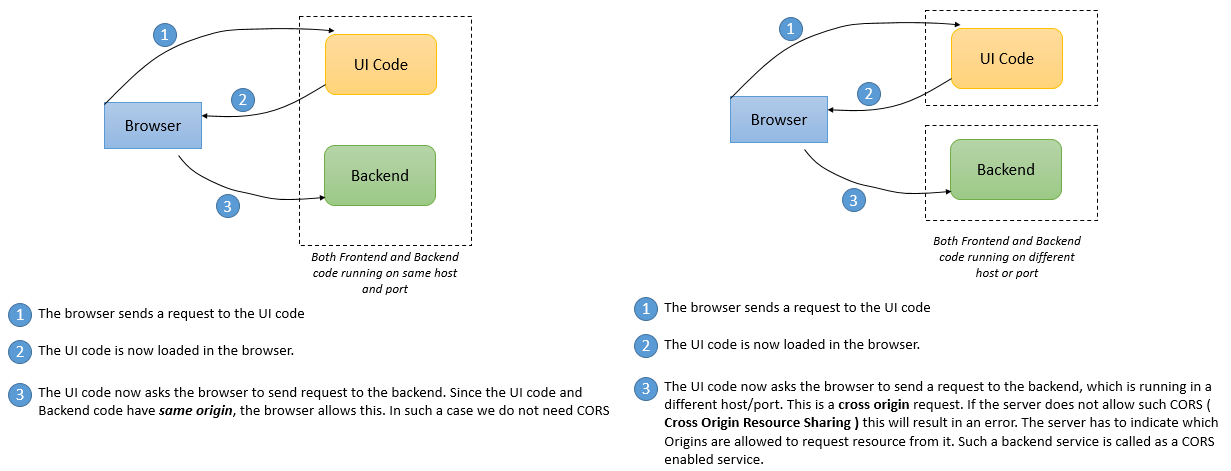
const bodyParser = require("body-parser");

app.use(bodyParser.json());

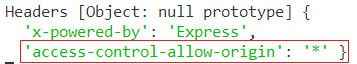
The incoming req.body will be in “{‘username’:’pavan’,’pass’:’123’}” so to make this string json file readable we use bodyparser.json() , this will make incoming req body string to json

CORS is Cross Origin Resource Sharing. Client is hosted on one server,backend server is hosted on another server, to allow interaction btw them is CORS. Web browsers only interact with same servers by CORS we can do this for different servers too. CORS adds HTTP headers which instruct browsers to allow cross-origin resource sharing. app.use(cors())

This is how res header will look like if req is made, Access-Control-Allow-Origin: \*



If CORS enabled



If CORS not enabled



# Promise

We create promise with new Promise keyword, it consists resolve, reject. Both are like response but resolve is successful, reject is like error. So if we have many callbacks then it will be callback hell which is not maintainable. To solve this we use Promises.

let isBalanceEnough = true;

let Bank = {};

Bank.checkPayerBalance = function() {

return new Promise((resolve, reject) => {

if(isBalanceEnough)

resolve('Checked Payer Balance,');

else {

reject('Not Enough Balance');

}

});};

Bank.deductAmountFromPayer = (message) => {

return new Promise((resolve, reject) => {

resolve(message + ' Amount Deducted From Payer,');

}); };

Bank.checkPayerBalance().then((result) => {

return Bank.deductAmountFromPayer(result);

}).then((result) => {

console.log('Finished, ' + result);

}).catch((err) => {

console.log(err + '\n Sorry, amount cannot be transferred!');

});

# Object Data Modelling

An ODM is used to map the data as JavaScript objects to the underlying database format. They provide an option to perform validation and checking data to ensure data integrity.

Mongoose is an asynchronous MongoDB environment in which we can do validations, constraints etc, Other ODM is mongo which doesn’t have any validations etc,

So in general, we create connection first, we create schema, we create model to do operations

npm i mongoose –save

const mongoose = require(‘mongoose’);

mongoose.Promise = global.Promise; //to deal with asynchronous code easily, configures mongoose to use global promise library

mongoose.set(“useCreateIndex”,true) //this is also like newURLParser:true in next step, to avoid deprecating error when we create index in schema

mongoose.connect('mongodb://localhost:27017/MultiplexDB', {useNewURLParser:true})// first parameter is URI of db, we can set to use new URL parser cause current one may deprecated in future which is optional

In mongoose, the structure of the data is defined in a schema.

const empSchema = new Schema ({

“name”: {string,required:true,default:”Unknown”,unique:true},

“empNo” : {type:Number,min:1,max:5},

"language": {

required: [true, 'Required field'],

  match: [/^[A-Za-z]{3,8}$/,'Please enter only alphabets'],

type: String,

index : true // creates index at field level

},

“address”:[{“doorNo”:string,validate : ()=>{//custom validate function}]

})

empSchema.index ({empNo:1}) //creates index at schema level

Let schema = mongoose.schema(empSchema,{collection : ‘Employee’’,timestamps:true}) //here we are assigning this schema represents collection employee, timestamps will add two new fields to collection which are



So we use this structure to create a mongo db model to connect with db and do operations. Model creates an instance of db with this schema.

Let model = mongoose.model(collectionName,SchemaName)

Let empmodel = mongoose.model(“Employee”,empSchema)

Empmodel.create({empobject}).then(insertedData)=>{return insertedData} //here we can use defined create etc or we do mongo operations directly like insert here. Empobject is the data {“name:..} to be inserted, another example is

Lets us now retrieve all the documents that we have inserted into our Movie Collection.

movie.find = () => {

return movieModel.find().then((movieData) => {

if (movieData) { return movieData }

else {

let err = new Error("No record found")

err.status = 404

throw err;

}

})

} // we can do moviewModel.insertMany([{},{}]).then(result)=>{…, so like this we can do find,insert,update,delete etc. .toArray() is used to send the result in form of array. Note: for update validators will not invoke so we have to mention: return movieModel.updateOne({movieId:2002},{$push:{showDetails:showObj}},{runValidators:true}).thn

To use mongodb instead of mongoose

Const mongo = require(‘mongodb’);

Const connection = mongo.connect(“mongodb://localhost:27017/<dbname>”)

In function return connection.then(mongoInstance)=>{

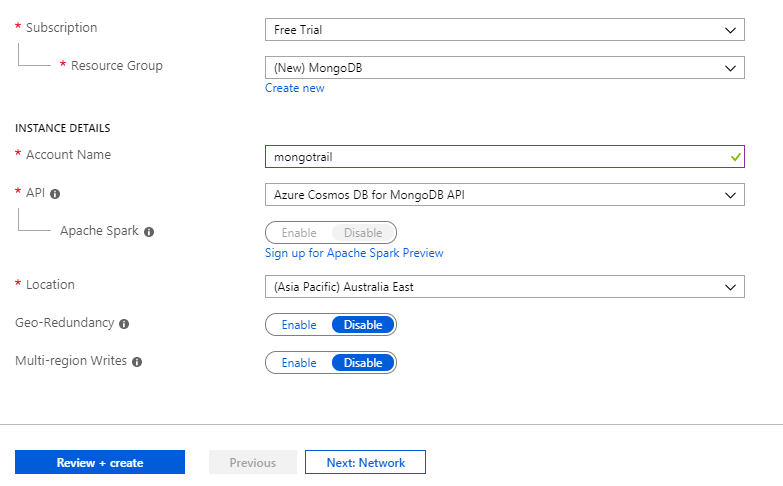
let database = mongoInstance.db();

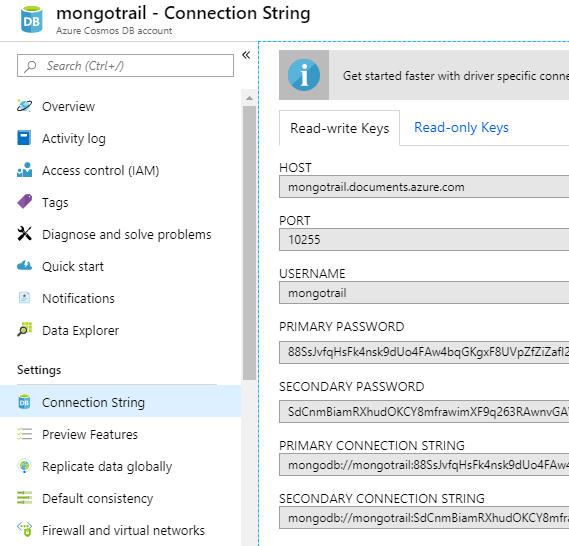
let collection = database.collection(‘Employee’)

return collection.find().toArray();

}

# Cloud





So we will create a environment.js file and maintain this data which is used to make connection,

In environment.js file

module.exports = {

accountName: username from connection string,

databaseName: 'EmployeeDB',

key: primary password from connection string, // replace all the "=" with "%3D" in your key

port: PORT from the connection string

};

In connection.js file

const mongoose = require("mongoose")

const connString = require("./environment")

const Schema = mongoose.Schema

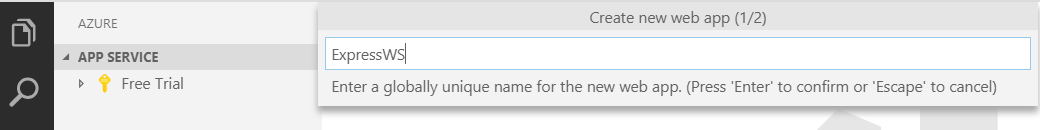
let connectionString = "mongodb://" + connString.accountName + ":" + connString.key + "@"

+ connString.accountName + ".documents.azure.com:"

+ connString.port + "/" + connString.databaseName + "?ssl=true"

app.listen(process.env.PORT || 3220); //to host server on cloud

In VS code, ad azure extension and then in that create a new webapp, select NodeLTS as linux environment



Then deploy our backend application in it by selecting the folder, after successful deployment if we hit the generated link it will give the response

# NoSQL Injections

In code we wrote Select \* from Employee where employeeId = + “inputdata”, then user gave inputdata 123 or 1=1 then query becomes Select \* from Employee where employeeId = “1 = 1 or 123” if this is typed in input box then it will result of all the data, to prevent these injection attacks we use validations of input (like here make sure it is number always, we do this in schema)and sanitize the input (to skip if input looks like a code).

const sanitize = require("mongo-sanitize");

so before we use req.body.employeeId directly in code we sanitize it and store in new empData

let empData = {

employeeId: sanitize(req.body.employeeId)

}; //in code we use empData.employeeId instead of req.body..

app.post('/getEmployee', (req, res) => {

let empData = {

employeeId: sanitize(req.body.employeeId)

};

console.log(empData);

conn.then((MongoClientInstance) => {

let database = MongoClientInstance.db();

let collection = database.collection('Employee');

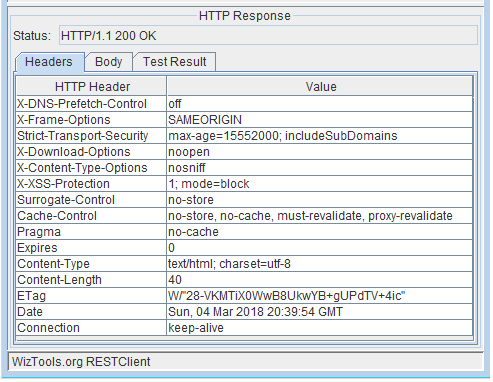
collection.findOne(empData).then(fetchedData => { console.log(fetchedData); }) });

To prevent clickjacking, script injection, vulnerabilities of client side caching we use helmet

Const helmet = require(‘helmet’);

app.use(helmet.noCache()) //disable client side caching

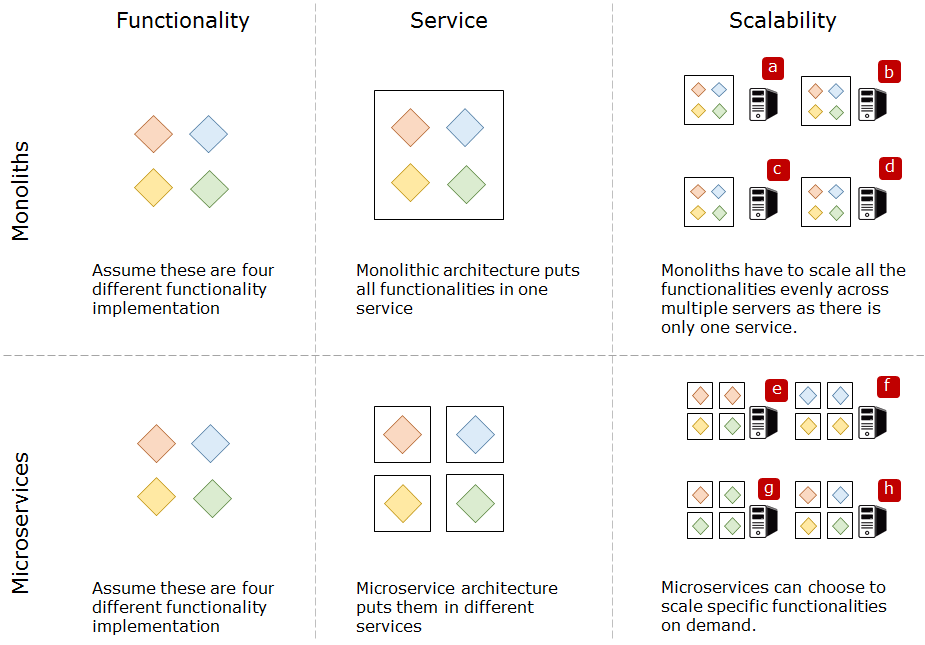
app.use(helmet.frameguard()) //to block clickjacking



# Microservices

Monolithic architecture means single server, single technology, large code at one place etc, it makes hard to maintain, large build and deploy times, stuck to one technology, had to scale entire application if one part faces more load etc, to overcome these we can use Microservice architecture.

In microservices we divide the large monolithic application or service into smaller bits or microservices with its own environment



In express we use hydra to create microservices, for express we use hydra-express. We use config file to enter details of that microservice like name,port etc. We use API gateway to let the communication btw them. In this we will get the express object from hydra-express not by the express directly as we did earlier, that is the only change here rest all are same. Create a config.json file manually in code and then

{

"hydra":{

"serviceName":"helloms",

"serviceIP":"",

"servicePort":8181,

"serviceType":"",

"serviceDescription":"Hello Microservice",

"redis":{

"url":"redis://127.0.0.1:6379/15" //start redis database and get url

}

}

}

npm i hydra-express;

this is routes.js

const he = require(‘hydra-express’);

const express = he.getExpress();

const router = express.Router();

router.get(‘/greeting’,(req,res,next) =>{

res.json({message:”Hello from MS”});

next();

})

This is app.js

const he = require(‘hydra-express’);

const express = he.getExpress();

const router1 = express.Router();

const router2 = require(‘./routes’)

he.init(‘./config.json’, () => {

router1.use(router2)

he.registerRoutes(‘/hello’:router1) //so if we hit localhost:8181/hello/greeting will get json.

}

So if we create another MS, its config file looks like for intercommunciation

{

"hydra":{

"serviceName":"wishesms",

"serviceIP":"",

"servicePort":8182,

"serviceType":"",

"serviceDescription":"Wish Microservice",

"redis":{

"url":"redis://127.0.0.1:6379/15" //start redis database and get url

}

}

}

this is routes.js

const he = require(‘hydra-express’);

const express = he.getExpress();

const router = express.Router();

const hydra = he .getHydra(); we will use hydra for intercommunication of MS

router.get(‘/birthday’,(req,res,next) =>{

const message = hydra.createUMFMessage({

to : ’helloms:[get]/hello/greeting’ //toAddress msname:[req method]path

from : ‘wishesms’

body : {} //useful in post request

})

//we created a message now to send it we create API request

hydra.makeAPIRequest(message).then(data => {console.log(data.message)})

next();

})

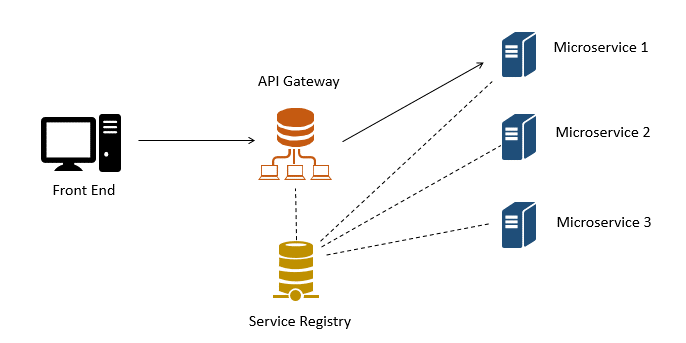
The app.js remains same we can change the registerRoutes path to ‘wish’ instead of hello

In the communication message, we didn’t mentioned port of where helloms is located but still it will hit the service cause both are in same redis server, they will communicate by ms name.

So if we hit localhost:8182/wish/birthday we can see “Hello from MS” in console.

If we remove default port numbers and make it to 0 then the MS will be hosted on dynamic ports(random available port), so if we create multiple instances of these, each have different ports. But each have unique instance ID which can be readable by hydra.getInstaceID(). We will get port details in our output terminal in vs code. So now link becomes localhost:1452 (random generated)/hello/greeting will give output json. If we create 3 instances like this and make a call from wish ms through message it will send request to one of the available helloms instance so automatically loabalanced.

Again, since it is random generated number we need one fixed port in the top of hierarchy which will store random ports and ms instances which is API gateway.



Service registry means redis database, API gateway is also another MS which is hosted on fixed port. This consists of other docker etc files so download it(hydra-Router API gateway) from the git and then in config.json give the fixed port number like 1111 and then redis url same as for other MS we used and run it. IF we hit localhost:1111 it will give dashboard consisting, how many ms are running their info like portnumbers, names, description etc, error & other logs,no of http requests, previous data and also its own api gateway details also. So to make a call we give path localhost:1111/helloms/hello/greeting gives the json output. First parameter is ms name and then path.

# TypeScript

Note: Js and Ts are case sensitive Var,Class etc will give error has to be var,class etc

To overcome dynamic typing(not setting datatype), minimal oop concepts like class,  encapsulation, interface, inheritance etc,no module support, Js is interpreted language so executes directly without compilation. We write code in ts file and convert them to js which is called transpilation.

tsc hello\_typescript.ts //creates .js file in same path; tsc hello\_typescript.ts –outdir JS //JS folder

tsc module1.ts module2.ts module1,2UsingFile.ts (tsc – typescript conversion)

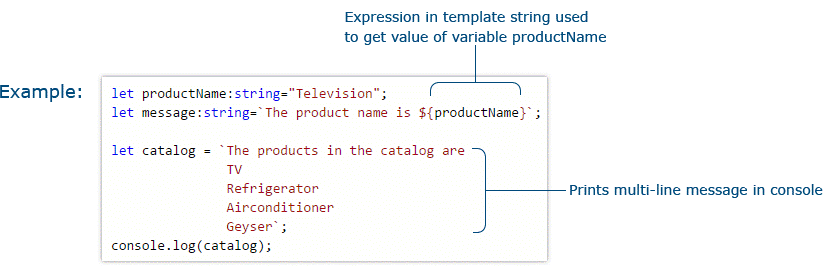
during modularizing the code, these become interdependent so has to transpile at once

tsc –sourceMap demo.ts // this will generate .js file as well as .map file. If we open this js file in browser, we can see ts file too in source tab of browser(go to the settings of the Chrome browser and check the option "Enable JavaScript source maps" of Sources). This how we link js and ts files and debug accordingly

let,const is a block scope, var is lexical scope

we can push,pop, add etc to **const** array but can’t reasssign entire array

In ‘’ we need to + for concatenation so we use ` ` supports concatenation by ${} and multiline.



Let name:any; takes any datatype; void is like nothing let n:void = undefined or null; if we have function which returns nothing then it is void type. Null is assigned if we don’t want any value to be assigned to it.

## Enum

If we want to store values with a names then we use enum, like for small,medium,large we will set values enum MobilePrice {Black=1500,White,Red=White+1000} by default it will take 0,1,2.. so here white values becomes 1501, red becomes 2501.

Can access like MobilePrice.Black or MobilePrice[“Black”] (if we have string)

for (let i in MobilePrice){ //of gives error

console.log(i) //gives 1500,1501,2501,Black,White,Red }

## Array

Let tempArr : string[]; let tempArr:Array<string>; for fixed length arrays and to take different datatypes unlike array which will take fixed data type, Typescript has tuples

Let tempTuple = [number,string,number] //it only takes 3 specified datatypes in same order and also unlike array which takes only string or number types, we can store mixed types in tuple. If we give more than 3 it will throw error because of Typescript data restriction policy, we can push multiple values instead like tempTuple.push(1,”a”,2,11,”b”,12,21,”c”,22) pushes into tuple successfully.

for (let i = 0; i < tempArr.length; i++) { //code implementation }

let i =0 ;for (;i< tempArr.length;){i++} //we can skip them if they are given somewhere else but need ;

for (let i in tempArr) {} // in is used so i gives indexes 0,1,2..

for (let i of tempArr) {} // of is used so i gives values of array directly here

## Function

function functionName(param1:Datatype, param2:Datatype):returnType {//body of the function }

By default all parameters are optional in JS, all are mandatory in TS. To take optional parameters we give ?, to make default we give value to parameter.

Function x(y=”defaultValue”,z?:number):string{ return y+z }

If we want to use default value then give like x(undefined,2) if default value is at last then we can directly give x(2), here z is optional. Default value cannot be an optional parameter. To take n no of parameters we use Rest Paramters which is denoted by 3 dots before, which it acts as array, and always should the last parameter. It should be declared as array always.

Function x (y:number, …restParams:string[]):void { for (let i of restParams){ console.log(i+y}}

If we invoke x (5,”Hi”,”Hello”,”Mi”) then output is Hi5 Hello5 Mi5

## Class

class Product{

prodId: number =1 //this will bind to this.prodId

getProd():number{ return this.prodId }

}

var x = new Product()

console.log (x.getProd())

If we want values form input then use constructor

We can either declare variables before constructor or within parameters of constructor

class Product{

private prodId : number = 1 //can give default value here

constructor (prodId:number) { this.prodId = prodId } // if we give without this like prodId = prodiId then this.prodId will refer to 1 default value cause we are not assigning inside constructor

instead we are assigning prodId input number to again same prodId variable so directly like below (inside constructor use this. Always else it won’t get assigned).

class Product{

constructor (private prodId: number) { this.prodId = prodId }

getProd():number{ return this.prodId }

}

var x = new Product(3)

console.log(x.getProd()) //3; if we print x.prodId gives error cause its private otherwise it gives 3

// private only with class, protected only with class and child classes, public anywhere

All these properties are bind to instance of class, we can give properties of class by static keyword

Static methods and static variables are binded to class not instance so has to call by using class name

class Product{

static count = 0

constructor (public prodId: number) { this.prodId = prodId }

getProd():number{ return this.prodId }

static getCou(){ Product.count++ ; console.log(Product.count) }

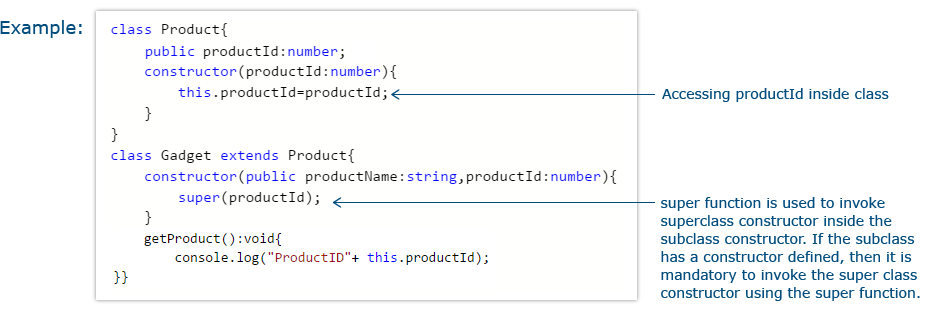
}

var x = new Product(3)

Product.getCou() //1

Product.getCou() //2

To make a child class or for inheritance we use **extends** keyword with super keyword for constructor



We can use super.getProductFunctions() if this function is in parent Product, we cannot access variables like super.productId gives error directly use this.productId only if it is protected or public

## Abstaract class

These are base classes which don’t have any implementation, if we want a class to have specified methods or properties always then we can define in abstract class and make our class extends it.

abstract class Product {

getCost(amount): void { //this function acts as parent class so we can call it with child class instance or super.getCost() in child class, no abstract keyword is mentioned so not considered.

console.log("The article costs Rs. " + amount)

}

abstract getProductName(): string; //only abstract keyword are considered mandatory

}

class Gadget extends Product {

getProductName(): string { //since this function is used, runs without error

return "ProductName is Mobile";

}

}

var g = new Gadget();

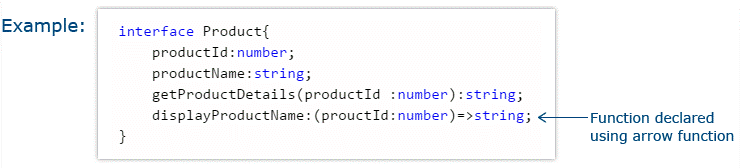
console.log(g.getProductName()); //"ProductName is Mobile”

var p = new Product(); // this will throw error cause abstract classes cannot be instantiated

g.getCost(1) // gives 1 output

## Interface

To maintain consistency, ensures proper values are passed, classes, functions etc that implement interface has to follow all mentioned. Should not mention access modifiers(private,public..) here. If not implemented mentioned then throws compilation error

interface Product{

productId:number;

productName:string; }

function getProductDetails(productobj:Product):string{

return "The product name is "+productobj.productName; }

let prodObject={productId:1001,productName:'Mobile'};

let productDetails:string=getProductDetails(prodObject);

interface Product{

x:string;

getProductDetails():string[];

displayProductName:(productId:number)=>string;

}

class Gadget implements Product{

x:string = "io"

getProductDetails(): string[]{

return ["Samsung","LG","Moto"];

}

displayProductName(productId:number):string{

if(productId==101)

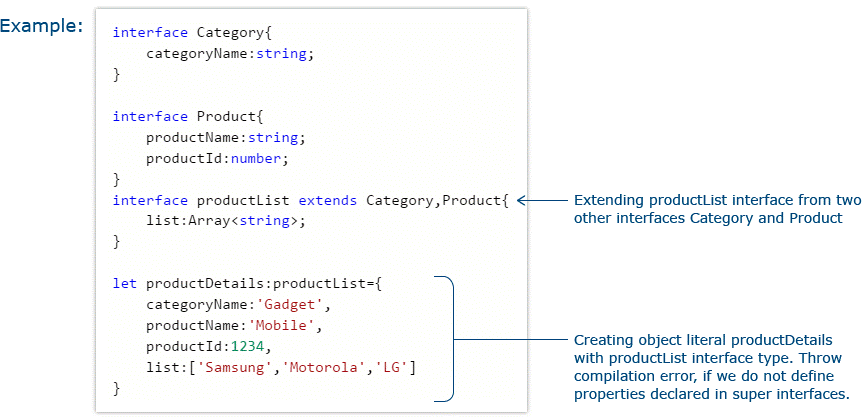
return "Product Name is Mobile";

}}

let g:Product=new Gadget();

let productName: string = g.displayProductName(101);

Like this we can combine many interfaces through implements keyword. To make few properties implement by child class then use Abstract with extend keyword, to define entire structure Interface



If alike properties then we use extends, if different then use implements keyword

## Modules

we can modularize code and store it in different files, we can export and import it. Code can be reusable and maintainable by this. We can export each property individually by adding export before it like export function x() { }; export class Product {},export interface y{},export const z…, or all at once like keeping all properties in one object and export that object directly module.exports = mainObject or we can export all collectively like export{x,Product,y,z as Zi}. we can export default but only one to be exported. export default class { } for importing we have to use same names of exported ones. If default class doesn’t have any name we can import with any name and should not be mentioned in {}, we can also use **as** keyword as alias in both exporting and importing

import defaultClass, {x, Product as p,y,Zi} from “./module1.ts”;

import \* as mainUtility from “./module1.ts” ; imports everything

## Generics

If we give fixed annotation :string,:number. We can only use for them. So to make this dynamic so that same fn can be used for different datatypes we use generics. <T> means Type parameter

Class Stack <T> { private data:T[]; push(item:T)=>data.push(item) }

let stackNum = new Stack<number>();

stackNum.push(0);

// stack.push("1"); // it will not be insert in stackNum as it is of a number type array

let stackStr = new Stack<string>();

stackStr.push("abc");

function x <T>.(data:T):T {return data} ;

Here, make sure that operations performed inside are common for all datatypes. We can’t do string operations on numbers or vice-versa so Generics will throw error if we do fixed datatype operation

Function y<T>(data:T) { data.substr(0,3} //this will throw compilation error cause .substr is only for strings not for other datatypes, here we need to directly specify string. Generics is only for common properties functions or classes etc like we did push into array above which is performed on all

## Decorators

Provides metadata, to specify behaviour, properties of the class or method. Used for declarative programming, implemented from angular 5

**Class Decorator**

Can define constructor of class, change class definition, actually this decorator will invoke a function at runtime with constructor of decorated class as its parameter

@Component, @NgModule, @Directive, @Pipe of Angular.

Method Decorator: @HostListener of Angular

**Property Decorator**

Can change, read etc. property value and decorator is applied to the variable representing property

@Input, @Output of Angular

## TSLint

Linting is used to do static analysis of code. It helps in early detection of errors, enforce uniform coding style and minimize the code size. (coding standards)

After the code is developed, we can lint it to check for programmatic and stylistic errors as well as non-adherence to coding standards and conventions. We can also remove unwanted spaces in the file or remove redundant code using a linting tool.

npm install -g tslint

tslint –init //this will generate the tslint.json file to specify the rules

{

"defaultSeverity": "error", // this is how tslint.json file looks, not important

"extends": [

"tslint:recommended"

],

"jsRules": {},

"rules": {},

"rulesDirectory": []

}

To specify rules we will define it in rules:{} key value pair

"rules": {

"no-var-keyword": true,

"semicolon": [true, "always", "ignore-interfaces"],

"class-name": true,

"no-console": [true, "log", "error"]

},

Once rules are specified, we can run linting test by following command

tslint -c tslint.json 'rootfolder/\*.ts //c means create (not sure), so we are asking to create tslint result based upon this tslint.json on this file of path, will get errors in VS code terminal

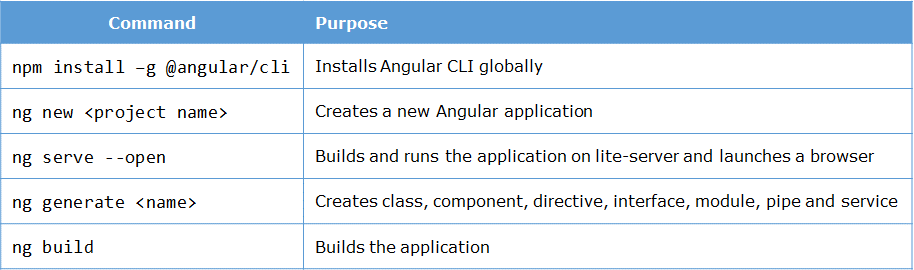
# Angular 8

Most powerful and performance efficient JavaScript framework mainly used to create responsive Single Page Applications (SPA) for both mobile and WEB, can create reusable components, pipes etc.

package.json will have all the dependencies, its versions which gets installed in node\_modules folder

tsconfig.json will have the ts files into js compiler options like outDir, sourceMap etc.

angular.json will have angular CLI related data like the packages we installed in node\_modules we specify there path here in angular.json (like defining bootstrap in scripts), index.html, css file paths



We can set port eg: ng serve --port 8183 –open or ng serve --open --port 3000

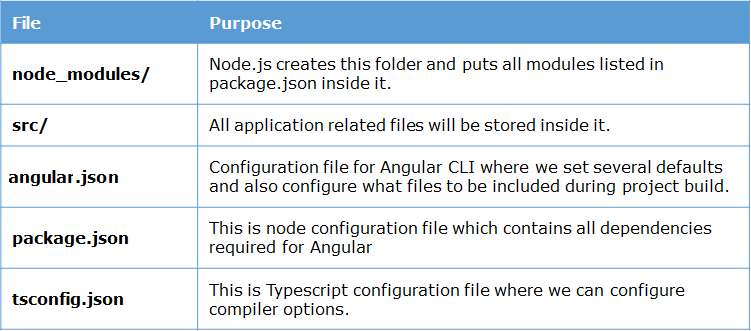
Angular uses MVVM principle i.e Model View Viewmodel, In model we define the structure (login.js) (interface) of enitity (just like schema model in mongoose), View has HTML templates, A viewmodel (.ts files) contains BL and gets connected with view thorugh data binding. Spec.ts consists test cases

class Login { //this is the model file of login consists

userName: string;

password: string;

}



tslint.json will have all the linting rules specified, e2e/ folder consists end to end test cases written in Jasmine and run by protractor.

## Bootstrap

npm install bootstrap –save

... // bootstrap is related to angular so we will add bootstrap.min.css in angular.json styles

"styles": [

"src/styles.css",

"node\_modules/bootstrap/dist/css/bootstrap.min.css"

],

...

## Component

Components are represented by defining a Typescript class with properties and functions and using a **special decorator**that makes them a component. The class interacts with the view through an API of properties and methods. Angular app is built component by component, Component is accessed by custom html tag with functionality (<app-root></app-root>).

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

@Component({

selector: 'app-root', // <app-root></app-root> // templateURL means external template

templateUrl: './app.component.html', //or template: `<h1>Inline template without URL tag</h1>`

styleUrls: ['./app.component.css'] //always place inside [ ] cause css can be multiple files

}) //for inline templates,styles use `` not ‘’ or “” also tag changes for both (styleURLs to styles and templateURL to template)

export class AppComponent { //the above @Componenet binds the below Appcomponent class with mentioned templates,styles,selector etc.

title:string = 'app'; // {{ title }} in html

constructor() { } //do everything inside this class

ngOnInit() { } // to do on initialization

}

## Modules

Angular is collection of modules, these are used to organize the angular app

A module in Angular is a class with **@NgModule** decorator added to it. @NgModule metadata will contain the declarations of components, pipes, directives, services which are to be used across the application. Every angular app has one root module which will load at first at launch of application

In **app.module.ts** (Root module) file placed under app folder, we have the following code

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

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

import { AppComponent } from './app.component'; //no need to mention .ts

@NgModule({ //defines metadata of module

declarations: [

AppComponent //all components,pipes, directives to load with application goes here

],

imports: [

BrowserModule //need browser module to run in browser, all modules goes here

],

providers: [], // services goes here

bootstrap: [AppComponent] }) // bootstrap declaration should contain component which component is to be loaded at start of application so root is placed generally , if it have other component then it will be rendered on load of apps

export class AppModule { }

In main.ts we will enable bootstrap, production mode etc (tells how application should behave).

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

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

import { AppModule } from './app/app.module';

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

if (environment.production) {

enableProdMode(); //will enable production mode which will run application faster

}

platformBrowserDynamic().bootstrapModule(AppModule); // method accepts name of root module as parameter which will load the given module i.e., AppModule after compilation with bootstrap properties.

## Template variables

<input #uName type=”text”> //like this we can create template variables (# means id)

<button (click) = “showUsername(uName.value)”> //if id=”uName” then uNme.value doesn’t work

Eg:

template: `<span>Welcome to {{dName}}</span>`, // dName = “India”

styles: ['./app.component.css', `#span{color:yellow}`, `.span{color:indigo}`]

Here the Welcome to India in browser displays in normal blackcolor without any styles because the tag styles: is used instead of styleURLs so the path is considered as string not url, 2 style is #span which indicates span id (#indicates ID) so there is no span id, 3 is .span which indicates span class. So none applies here, to apply we have to use tag here which is <span> { color:’blue’ }.

## Data Binding & Interpolation

Suppose there is title which is a variable inside the appcomponent class, we can use it in html using interpolation {{ title }}, we can also do operations, {{ title.subtsr(1,2) }}, to avoid error if title is undefined we give {{ title?.subtsr(1,2) }} , we can also give our own {{ ‘hello’.substr(2,3) }}. This method of using variables in viewmodel in template by {{ }} is called Interpolation.

If we want action to be from template to Viewmodel(class) then we use Event binding (event)=statement. Eg: (click)=”change()” //on click in template it will invoke change() in model, from viewmodel to template we can use []. Two way [(…)]. We can also use bind,on keyword as mentioned in below pic

For one way data binding we can do (property means necessary like length breadth etc. of input, attribute means its behaviour which are not necessary like placeholder of input

Property Binding, Attribute Binding, Class Binding, Style Binding, Event Binding (click etc.).

Property binding:

<img [src] = 'imageUrl' /> or <img bind-src = 'imageUrl' /> or <img src = “{{imageUrl}}”/>

We can also pass it as string still it refers same variable eg: src = {{“imageUrl” }} or [src]=”’imgUrl’”

In .ts file component class, imageUrl:string = ‘.assets/imgs/img1.jpg’;

Property binding only binds property, for other bindings we have to mention it before like attr. etc

Note: Should not use bindings & interpolation together eg:<img [src] = “{{imgUrl }}” >doesn’t work.

**Attribute Binding:**

Attribute bindings are used to change appearance and behaviour of DOM element. Eg: for a table no.of rows, columns are properties. But the properties of these rows, cols like colspan, rowspan etc. are attributes. We cannot do interpolation or property binding to attributes so <td colspan = {{colValue}} /td> (for input field, width height are properties, placeholder is attribute (add on))

This will give error, so use attr. Eg: <td [attr.colspan] = “colValue” /td>.

**Class Binding:**

Same like attribute, class binding also can be done only by mentioning [class.className]

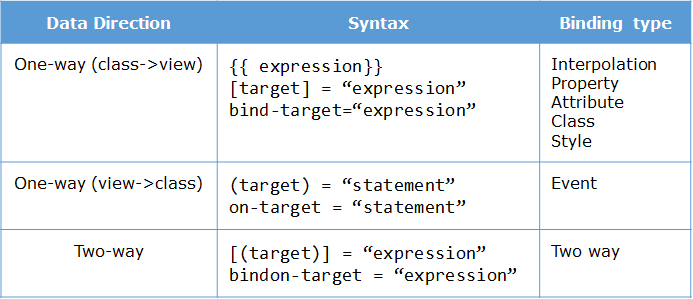
<div [class.mycss] = “isValid” ></div> //if isValid is true then mycss class will apply, in general we use ngClass module cause it can apply multiple classes.

<p **[ngClass]='["pink"]'**> background is pink </p> //here we are giving array of classes to be applied, so it will work. (imp syntax) applied when there are multiple values.

**Style Binding:**

<button [style.color] = "isValid ? 'blue' : 'red' ">Hello</button>

<button [style.font-size.px] = "isValid ? 3 : 6">Hello</button> //ngStyle to apply multiple styles



<input [(ngModel)] = "course.courseName"> // this equals to below line

<input [ngModel]="course.courseName" (ngModelChange)="course.courseName=$event">

// ngModel means that tag value attribute, $event gives the changed value

To make two way data binding i.e., [(ngModel)] work, import FormsModule class to the root module as shown below

@NgModule({

imports: [

BrowserModule,

FormsModule

],

...

})

## Directives

One which alters the view is called directive.

ngIf, ngFor, ngSwitch,ngStyle,ngModel,ngForm etc. Directives are used to change the DOM. @Component is a component directive which alters the HTML, ngIf,ngFor,ngSwitch .. these are structural directives which can alter structure of html. Attribute directives are ngStyle, ngClass etc. We use\* before like \*ngIf = expression, \*ngFor = expression etc. in html(view).

\*ngIf doesn’t hide the element instead it removes the element from DOM tree. So less resources, high performance because hidden elements still listens to events, this is the advantage of directives.

<div \*ngIf = “isTrue” > </div> //if isTrue is true then this div tag is considered.

<ng-template> has to used with structural directives only, can be used instead of \*ngIf for example

<div \*ngIf="submitted">

<div \*ngIf="isAuthenticated; else failureMsg">

<h4> Welcome {{userName}} </h4>

</div>

<ng-template #failureMsg> //it equals to <div \*ngIf=”!isAuthenticate”>

<h4> Invalid Login !!! Please try again...</h4>

</ng-template>

<button type="button" (click)="submitted=false">Back</button>

</div>

\*ngFor – for iterating the list

<ul>

<li \*ngFor="let course of courses; let i = index"> //courses = [{name:”s”},{..} ..]

{{i}} - {{ course.name }}

</li>

</ul>

\*ngSwitch

...

<div [ngSwitch]="value">//unlike JS switch it keeps on going all cases till end even one case matches

<p \*ngSwitchCase="1">First Choice</p>

<p \*ngSwitchCase="2">Second Choice</p>

<p \*ngSwitchCase="3">Third Choice</p>

<p \*ngSwitchCase="2">Second Choice Again</p>//if value is “2” then this <p> visible not 2 <p>

<p \*ngSwitchDefault>Default Choice</p>

</div>

...

ngStyle

<p [ngStyle]="{

color:colorName,

'font-weight':fontWeight,

borderBottom: borderStyle

}">

Demo for attribute directive ngStyle

</p>

ngClass //like others, it will also dynamically set and changes css classes

<div [ngClass]="{bordered: isBordered, myColor: isColor}">

Border {{ isBordered ? "ON" : "OFF" }}

</div> //isBordered,isColor = true in .ts file or component class

Add the following CSS classes in **app.component.css**

.bordered { .myColor{ color:blue }

border: 1px dashed black;

background-color: #eee;

}

### Custom Directives

All tags like input,required, everything we write in html is a default directive, We can create custom directives and we can use them throughout the application like for example if we need an email validation generally we validate inside form, but if we create it as custom directive we can use same validator logic in entire app across components. (we created autofocus directive an used it)

ng g directive EmailValidator

import { Directive } from '@angular/core'; //automatically comes

import { NG\_VALIDATORS, FormControl, Validator } from '@angular/forms'; //since we validate form

@Directive({ import these

selector : ‘[emailValidate]’; // we use this selector whenever we need inside tags to implement this logic <input type= “email” emailValidate > this will work as emailValidate is a property now

providers: [

{ provide: NG\_VALIDATORS, useExisting: EmailValidator, multi: true }

] // provide default ng\_validators(all default like required,maxlength will be in here) and add our

}) class EmailValidator to this by giving multi : true so that we can use this wherever just like default

export class EmailValidator implements Validator { //Validator interface implements validate()

validate(control: FormControl) {

const emailRegexp = /^[\w.\_]+@[A-Za-z]+\.(com|co\.in|org)$/;

if (!emailRegexp.test(control.value)) { return { "emailValid": true };

}else {return null} } //just like in model driven form validator send null if error not persists.

To create a simple understandable directive, if we want to add lines inside below tag

<span changeText >Welcome to Angular.</span> // we create changeText directive

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

@Directive({

selector: '[changeText]' // chnageText is a property so kept inside []

})

export class ChangeTextDirective {

constructor(Element: ElementRef) { //since this is not reactive form we are not using formcontrol

console.log(Element); so directly referring by elementref

Element.nativeElement.innerText="Text is changed by changeText Directive. ";

}

} Declare this in declarations of app.module.ts to use in entire application

Output: Text is changed by changeText Directive. Welcome to Angular.

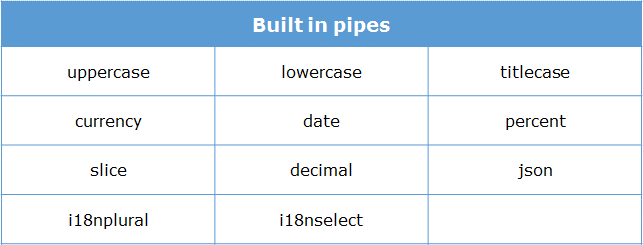
**<ng-content></ng-content>** this acts as a placeholder for the content inside the tag

Eg: @component{ selector :”add-btn”;template:”<button….><ng-content></ng-content</button>” This is how a component is created so that to make reusable code of button, when we use this inside other component like : ……<add-btn >Sign Up</add-btn> now button comes and in the place of ng-content tag the Sign-Up comes so a sign-up button comes. Its an equal to sending text to child component and then using that as interpolation in it, we avoided these 2 steps by ng-content.

## Pipes

Pipes are used to format the data before displaying it to the user. A pipe takes data as input and transforms it into the desired output.

{{ expression | pipe }} like {{ "Technology" | uppercase }}



A pipe can also have optional parameters to change the output. To pass parameters, after a pipe name add a colon( : ) followed by the parameter value.

Eg:

{{ 66600 | currency :'INR' :'code' :'1.3-4'}} <!-- will display INR66,600.000 --> <br>

Code indicates take ‘INR’ as code, 1.3-4 means {minIntegerDigits}.{minFractionDigits} - {maxFractionDigits}

{{ 0.1 | percent:'2.2-3' }} will display 10.00% , {{ 0.1 | percent }} will display 10%, converts 0.1\*100%

{{ 25000 | number:'.3-5' }} will display 25,000.000

{{ ['a','b','c','d']| slice:1:3}} will display b,c

{{ 'Laptop Charger'| slice:-4}} will display rger

{{ "6/2/2017" | date }} will display Jun 2, 2017

{{ "6/2/2017, 11:30:45 AM" | date:'medium' }} will display Jun 2, 2017, 11:30:45 AM

{{ "6/2/2017, 11:30:45 AM" | date:'mmss' }} will display 3045 //only minutes and seconds

{{ 90000000 | date }} will display Jan 2, 1970, date pipe will start from Jan 1, 1970 and count milliseconds from that date

shortDate' : equivalent to 'M/d/yy' (e.g. 1/31/18)

'mediumTime' : equivalent to 'h:mm:ss a' (e.g. 11:05:04 AM)

'fullTime': equivalent to 'h:mm:ss a zzzz' (e.g. 11:05:04 AM GMT+05:30)

'full': equivalent to 'EEEE, MMMM d, y, h:mm:ss a zzzz' (e.g. Wednesday, January 31, 2018 at 11:05:04 AM GMT+05:30)

{{ {'productId':1234, 'productName':'Samsung Mobile'} | json }} will display {"productId":1234, "productName":"Samsung Mobile"}

<p> Month is{{ "2018-11-11" | date | uppercase | slice:0:4 }} </p> //chaining of pipes

**Custom Pipes:**

If we want to implement sorting, filtering etc we will use custom pipes

ng g p salutation(pipename) //this creates salutation.pipe.ts and .spec.ts for test cases

**PipeTransform**interface has transform() method where we need to write custom pipe functionality, it takes 2 values one is expression and 2 one is the various optional parameters like we saw in built in pipes(since 2 parameter can be many and not fixed so this can be rest parameter, if fixed no need).

So we write our class implements pipetransform and write the logic in transform function.

@Pipe({

name: 'salutation'

})

export class SalutationPipe implements PipeTransform {

transform(value: any, args?: any): any { // fixed only one argument so no need of else …args?:any[]

if(args == 'M'){

return "Mr."+value;

}

else if(args == 'F'){

return "Ms."+value;

}

else{

return value

} } }

{{ username | salutation : 'M' }} //username=Pavan gives Mr.Pavan output

No need to import this in the component .ts file, to make this pipe available throughout the application we will declare it in app.module.ts

import { SalutationPipe } from './salutation.pipe'; Inside declarations: where we declare components

## Nested components

If we load a component inside another component then nested components. Parent component is also called as container component. We call it by selector inside html.

**Component communication from container/parent to child component**

In general here, img is also a kind of child component which are built in

<img [src]="iconUrl"/> // src and click are properties belonged to template, iconUrl, onSave()

<button (click)="onSave()">Save</button> /belong to class. Both are binded together as component.

So same applies here, if we want to bind the child component property with parent class variable:

<app-book [bookToDisplay]="userBook" ></app-book>

We need to tell angular that bookToDisplay is property of app-book component which will taken from parent/container component. To make this we use @Input decorator in child’s class

The property of a component is considered as being private for binding with the property of other components. By writing **@Input()** decorator, we are making the property as public for binding from Angular perspective. After the use of the decorator, that property can receive input from any other components or directives.

So the input decorator should be used only when the binding is between 2 different component's properties.

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

...

export class BookComponent {

@Input()

bookToDisplay: string;

}

The only way to pass the data from child to parent is through event emitter by @Output decorator

In parent we call child and invoke our function to capture event or data from the child componenet

<app-login (customevent)="updateFromChild($event)" ></app-login> //in parent template

…

export class AppComponent { //in parent class

count: number = 0;

updateFromChild($event){

this.count = $event;

}

}

<button (click)="emitevent()">Event emitter</button> //In child template we invoke emitevent()

...

export class LoginComponent {

@Output()

customevent: EventEmitter<number> = new EventEmitter<number>();

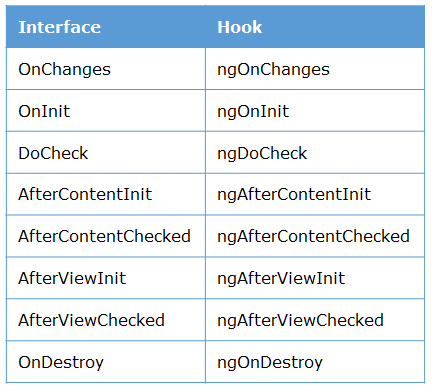
emitevent(){

this.customevent.emit(11);

}

} //In child class, we use @Output and then to pass data we created a customevent which is a EventEmitter<number> type and then we use this to pass 11 value inside eventemit function, since customevent is emmited in our parent updateFromChild($event) this is invoked and makes count 11

## Life cycles



**ngOnInit** – It will be invoked when Angular initializes the directive or component. This will be called **after**the constructor(). Typically constructor should not have any code and the setup code should be inside ngOnInit(). (imp point)

**ngOnDestroy** – It will be invoked before Angular destroys directive or component

Has to be imported and these are interfaces so has to be implemented

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

...

export class AppComponent implements OnInit, OnDestroy { //Inside OnInit interface ngOnInit class

ngOnInit() { }

ngOnDestroy() { } }

## Services

A component class is responsible for managing view and its logic, we should not implement data related code in it cause angular follows **Single Responsibility Principle(**programming principle that states that every module or class should have responsibility of a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class

**)** so we write all the data managing logics in services, Services can be used to make **HTTP** requests.

Service is a normal class with Injectable decorator to inject into component, if we create a class and use the data in it by creating a new instance of that class then it is a service. Like (ng g service book)

export class BookService {

constructor() { }

getAllBooks():Book[] { // Returns an Array of types

return [

{"id": 1, "name": "HTML 5" },

{"id": 2, "name": "CSS 3" },

{"id": 3, "name": "Java Script" },

{"id": 4, "name": "Ajax Programming" }

];

}

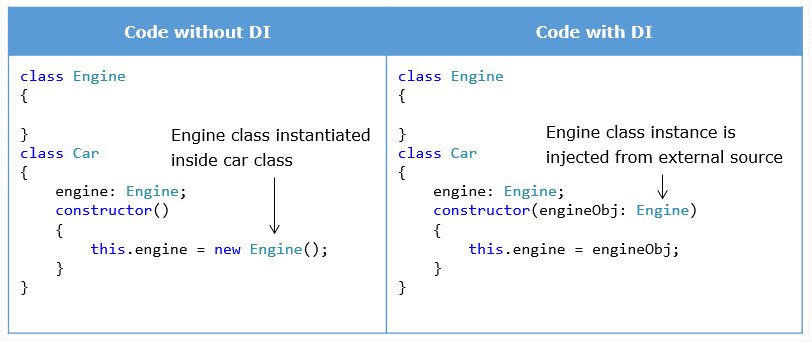
ngOnInit() {

this.bookService = new BookService(); // Creation of BookService instance using new keyword

this.bookListFromService = this.bookService.getAllBooks(); // Invoking the getAllBooks() of bookService to get data

}

The problem here is we are creating instance new BookService() everytime we need service class, but later if we make any changes like adding a new variable inside constructor then we have to change all these instances and have to pass the value. Also, unit testing will be difficult because we need to create a mock service instance or object to test it. To overcome all these instead of creating a new instance we will add that object through **dependency Injection**. This will allow us to use the code without creating an instance by passing it as parameter instead

So after creating the bookservice, we will inject the dependency like

@Component({...})

export class BookComponent implements OnInit {

bookListFromService:Book[]; // Will store the bookList to be displayed

constructor(private bookService:BookService) { } // bookService instance passed in constructor

ngOnInit() {this.bookListFromService = this.bookService.getAllBooks(); // invoking the getAllBooks() of bookService to get data } }

To make this service available to entire application then add it in providers of app.module.ts

@NgModule({

declarations: [ AppComponent, BookComponent ],

imports: [ BrowserModule ],

providers: [ BookService ], // we will add in providers whenever there is injection dependency fetch

bootstrap: [AppComponent]

})

export class AppModule { } If service is needed only for specific component(s), it can be added in those component classes instead of module.

### HttpClient

import {HttpClientModule} from ‘@angular/common/http’; add in imports of app.module.ts

we will pass this in constructor like Injection dependency, All http requests are asynchronous and An observable object can be used to represent asynchronous requests. An Observable is a 'collection that arrives over time'. Observable produces data which a subscriber can subscribe to and then use it. (Just like how we subscribe to newspaper and read it daily). In Angular, observables are handled using a third party library called **RxJs Observables**. Even Promises also do the same thing, handle asynchronous requests but we don’t use them cause 1) We can cancel observable if we don’t want response but in promise we can’t it will resolve or reject. 2) Promises emit single value, observable emit stream of values 3) Observable support functional operators like map, reduce, filter etc.

**Note:**We can convert any data into an observable using the **of()** method. For example,

import {Observable, of} from 'rxjs';

myObservable:Observable<number> = of(1, 2, 3);

myObservable2:Observable<number[]>=of([1,2,3],[4,5,6],[7,8,9]);

display(){ //Take this as a glimpse of testing a service, briefly mentioned in Testing below

this.myObservable.subscribe(

data=>console.log(data)

)

this.myObservable2.subscribe(

data=>console.log(data)

)

}

fetchData() {

this.data = new Observable(observer => {

setTimeout(() => { observer.next(11); }, 1000),

setTimeout(() => { observer.next(12); }, 2000),

setTimeout(() => { observer.complete(); }, 3000)

});

let sub = this.data.subscribe((value) => this.myArray.push(value),

error => this.errors = true,

() => this.finished = true);

ng generate service Book

@Injectable({

providedIn: 'root',

})

export class BookService {

constructor() { } } // Here, @Injectable indicates that the service may have its own dependencies and makes class as injectable. The providedIn: 'root' indicates that this should be available to all the components of the module. This is optional if we included it in providers array of app.module.ts

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

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

import { Observable } from 'rxjs';

import { Book } from './book';

@Injectable()

export class BookService {

private booksUrl = './assets/books.json';

constructor(private http: HttpClient) { } //Injecting http

getBooks(): Observable<Book[]> {

return this.http.get<Book[]>(

this.booksUrl

); } } // get(): Observable<any> { return this.http.get<any>( this.Url ); }

To retry the URL multiple time we use retry function and combine it with get method by pipe()

import { retry } from 'rxjs/operators';

getBooks(): Observable<Book[]> {

return this.http.get<Book[]>(

this.booksUrl

).pipe(

retry(3)); //3 time retry happens only if request made above is not successful

}

}

Pipes lets us combine multiple functions into a single function. The pipe() function takes as its arguments the functions we want to combine, and returns a new function that, when executed, runs the composed functions in sequence.

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

import { map, catchError } from 'rxjs/operators';

constructor(private http: HttpClient) {

if (sessionStorage.getItem('selectedProducts')) {

this.selectedProducts = JSON.parse(sessionStorage.getItem('selectedProducts'));

}

}

getProducts(): Observable<Product[]> {

if (this.producttype === 'tablet') {

return this.http.get<Product[]>('assets/products/tablets.json').pipe(

catchError(this.handleError));

} else if (this.producttype === 'mobile') {

return this.http.get<Product[]>('assets/products/mobiles.json').pipe(

catchError(this.handleError));

}

}

getProduct(id: number): Observable<Product> {

return this.getProducts().pipe(

map(products => products.filter(product => product.productId === id)[0]));

}

private handleError(err: HttpErrorResponse) {

console.log(err);

return Observable.throw(err.error() || 'Server error');

}

}

## Forms

* **Template driven forms** - Used to create small to medium sized forms
* **Model driven forms or Reactive forms** - Used to create large forms

**Template driven forms :** databinding, validations etc. will be written inside template. Template-driven forms use ngForm and ngModel directives to get information about the form and its controls and ngSubmit event to submit the forms.

* **ngForm:** Provides information about the current state of the form including a JSON representation of the form value and the validity state of the entire form
* **ngModel:** Provides two way data binding between the view and component. It is also used to track the state and validity of the input field.
* **ngSubmit:** Fires an event specified by ngSubmit when the form is submitted

To make all the above form directives, we need to import FormsModule into imports

<form #formRef = "ngForm" (ngSubmit) = "onSubmit()">

<div class="form-group">

<label for="idname">User Name:</label>

<input class="form-control" name="controlname" [(ngModel)]="classVariable" # controlname ="ngModel" id = “idname” minlength = “5”> //name in this tag is will be key value of form json

<div \*ngIf = “controlname.invalid && controlname.touched”>Username is wrong</div> </div>

</form> // formRef is template variable which is ngForm instance and represents entire form, classVariable is ts file variable wheare as #variable (optional since we gave #formRef entirely) is for template which captures instance of that tag ngModel mainly used for validations. formRef.value will give the entire form in json value with keys as name field of that tag an value entered.

You need a template reference variable to access the input box's Angular control from within the template. Here you created a variable called name and gave it the value "ngModel".

Applicable only to small form pages, more tags means more databindings and complex code. Cannot perform unit testing on form validation logic, only way is to perform end to end testing in browser.

**Reactive or Model driven forms:** This is a reactive type form,We create form control objects in component and binds them in HTML elements, main advantage is both value updates and validations are synchronous. Can use built-in and custom validators. We use FormBuilder for creating this and we need to mention **ReactiveFormsModule** in imports of app.module.ts

import { ReactiveFormsModule } from '@angular/forms';

@NgModule({

....

imports: [

ReactiveFormsModule

],

....

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

export class RegistrationFormComponent implements OnInit {

registerForm: FormGroup; // we create form group type property

constructor(private formBuilder: FormBuilder) { }

ngOnInit() {

this.registerForm = this.formBuilder.group({ // formbuiler.group() creates formgroup, it takes

formControl names and its definations

firstName: ['', Validators.required], //[default value, Validators]

lastName: ['', Validators.required], //[' ', [Validators.required,Validators.maxlength(10)]]

address: this.formBuilder.group({ // Note: address is another formgroup here

street: [], // we can also create like street: new FormControl(null)

city: [],

email:[‘’,ValidateEmail] // this is a custom validator (discussed at last)

}, {updateOn: ‘blur’)) // if we mention this in the end then validations occurs after focus is

} removed from that field else it happens on every keypress

} // so in html we give formControlName to bind form controls mentioned above

<form [formGroup]="registerForm" (ngSubmit)="submitted=true">

<div class="form-group">

<label>First Name</label>

<input type="text" class="form-control" formControlName="firstName">

<p \*ngIf="registerForm.controls.firstName.errors" class="alert alert-danger">This field is required!</p>

</div> //registerForm.get(‘firstName’).value gives value, we can set value like registerForm.setValue({firstName:”John”}), we can also do addControl(), removeControl(), reset() makes form pristine and untouched to starting and values to null if not mentioned any inside reset. contains(controlName) gives Boolean true or false if that control name contains in formgroup, this.registerForm.markAsDirty() makes the form dirty and same applies for markAsPristine etc.

<https://angular.io/api/forms/FormGroup>

Now, we can perform validations unit testing(Jasmine), can listen formControl and formGroup changes easily. We use this form for medium and large scale applications.

function validateEmail(c: FormControl) {

let EMAIL\_REGEXP = /^([a-zA-Z0-9\_\-\.]+)@([a-zA-Z0-9\_\-\.]+)\.([a-zA-Z]{2,5})$/;

return EMAIL\_REGEXP.test(c.value) ? null : {

emailError: {

message: "Email is invalid" //has to return null if it is success other than anything null it

} considers as validation failed, we can use this error message

}; to display in template

}

Can bind with built in validator like email: ['',[Validators.required, validateEmail] ]

<p \*ngIf="registerForm.controls.email.errors.emailError" class="alert alert-danger" > registerForm.controls.email.errors.emailError.message</p> </div>

For post request we use eg: sendData(this.registerForm.value) this sends form as json file

sendData(data: any) : Observable<any> {

return <Observable<any>> this.http.post("http://localhost:5400/login",data) ;

}

sendData(this.loginForm.value)

.subscribe(

(success) => { this.successMessage = success.message },

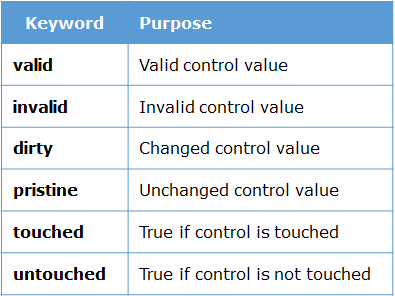
(error) => { this.errorMessage = error.error.message }

);

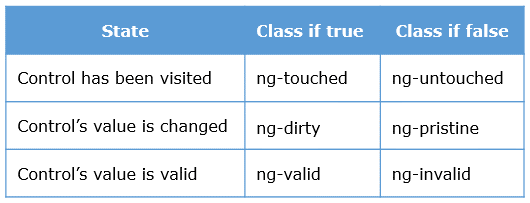
To boost the performance we use updateOn in formgroup as discussed above, by default is change (validations occur on change of form values), blur is discussed above, updateOn:’submit’ is another way. By blur and submit we can skip validations for every small change.

## State

Available keywords to track control state.



 built-in CSS classes to change the appearance of the control based on its state



In .css

.ng-valid[required] {

border-left: 5px solid #42A948; /\* green \*/

}

.ng-invalid:not(form) {

border-left: 5px solid #a94442; /\* red \*/

}

## Routing

Just like in express app.js we configure roots by app.use(‘/’,router), In angular also in root file i.e. index.html in <head> tag by default it will generate <base href =“/”> this represents the root path. So if we give /home inside base tag out entire application routes work from /home only.

For routing we need routing module and routes, both from “angular/router”;

ng g module app-routing –routing; or during creating app we can tell what we need like

ng new myApp –routing –style=scss ,creates routing file app-routing.moule.ts, changes the style in component styleUrls: ['./app.component.scss'] (scss instead of css] etc. can be done

import { RouterModule, Routes } from '@angular/router'; //In app-routing.module.ts

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

import { BookComponent } from './book/book.component';

const appRoutes: Routes = [

{ path: '', redirectTo: '/login', pathMatch: 'full' }, //pathMatch should be given when redirect is used

{ path: 'login', component: LoginComponent }, telling redirect only if path matches fully

{ path: 'bookDetails/:bookId', component: BookDetailsComponent } //bookId is params

];

@NgModule({ //all these are by default generated but we should know the reason

imports: [RouterModule.forRoot(appRoutes)],

exports: [RouterModule]

})

export class AppRoutingModule { } //declare this in import array of app.module.ts

After creating appRoutes we need to tell use this routes from root path (base href). So we are importing RouterModule adding our routes and exporting it. In app.module.ts we will add this updated routerModule in imports and use it in entire application. To use it in template

The **difference**between href and routerLink is that **href**will reload the page, whereas **routerLink** will not reload the page.

RouterLink - binds the router to links on a page and navigates to the appropriate application view when the user clicks a link.

RouterOutlet - used to specify where the views produced by the router is to be displayed.

If we want to pass it as single link the routerLink = “/booklist/1”, if separate then [routerLink] = “[‘/’,’bookList’,’1’]” (anything works) if we give ../booklist then it will move one parameter ahead

<nav>

<a [routerLink]='["/bookList"]'>My Books</a> //clicking on MyBooks will go to linked bookList

</nav> component and loaded in router outlet tag place

<router-outlet></router-outlet>

If there are multiple router-outlets then we can name them and differentiate (**auxiliary** route)

<nav>

<a [routerLink]='["/bookList"]'>My Books</a>

<a [routerLink]='["/bookAuth"]'>Authors</a>

</nav>

<router-outlet name = “list”></router-outlet> //after giving unique names to outlets

<router-outlet name = “auth”></router-outlet>

In our app-routing.moule.ts where we configure routes we give this outlet name

Const appRoutes:Routes = [ //path: ‘\*\*’ matches everything so has to be last

{path:’/bookList’, component:BookListComponent, outlet:”list”}, load in list outlet

{path:’/bookAuth’, component:BookAuthorComponent, outlet:”auth”} ,data:{title:’hi’}]. Pass data

const routes: Routes = [

{ path: "products", component: ProductListComponent },

{ path: "product/:id", component: ProductDetailComponent },

{

path: "",

component: SidebarComponent,

outlet: "sidebar"

},

{

path: "products",

component: ProductListSidebarComponent,

outlet: "sidebar"

}

];

In template:

<a [routerLink]="[{ outlets: { primary: ['products'],sidebar: ['products'] } }]">

Products List

</a>

<router-outlet></router-outlet>

<router-outlet name="sidebar"></router-outlet>

Here, in routerLink we are saying load products router output in primary outlet (primary means without any name has to be one for template) and load products again in sidebar outlet. In router first one matches and load prouctlistcompoenent in primary outlet, second searches with matching path and outlet name which is last one and loads sidebarcomponent. So this is how we can tell in routerLink that which part should display in which router outlet . no need to give same path we can invoke different paths also like “[{outelts: {primary:[‘products’],sidebar:[‘anotherPath’]}]” this will load anotherPath matching and sidebar outlet matching component in sidebar outlet. This is called Aux or Auxiliary Route (means routing by naming.

In code navigating can be done like **this.router.navigate**([‘/bookList’]) , navigateByUrl()

Import {Router} from ’@angular/router’, inject in constructor (private router:Router).

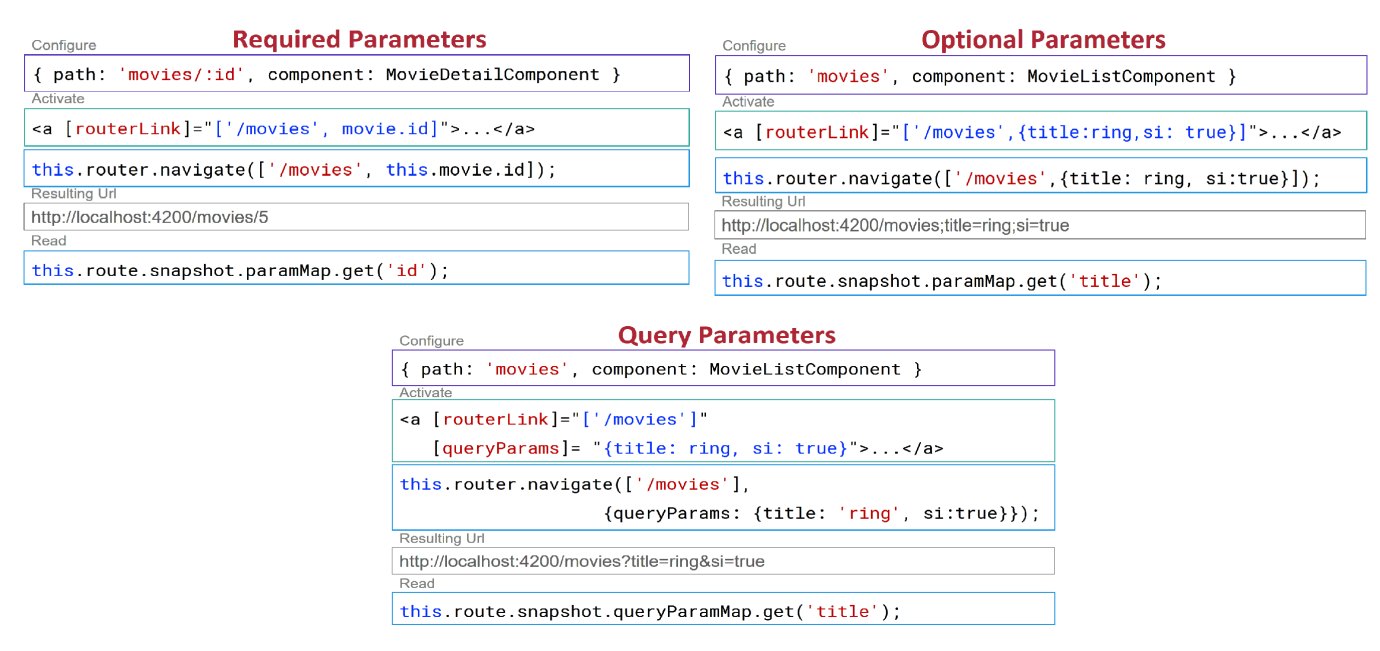
Syntax : this.router.navigate([url, parameters])

To get parameters from url we need **ActivatedRoute** this gives details of current activated route:

import {ActivatedRoute} from ‘@angular/router’;

constructor( private route: ActivatedRoute)

this.route.params.subscribe(param => this.bookId = param['id']); or can use snapshot doesn’t matter



To pass the data like post, since this is not a http request this is just based on paths we create a state object and pass the data like we do for body in post. Applicable from 7.2+ angular version

In home component

import { Router, NavigationExtras } from '@angular/router';

const navigationExtras: NavigationExtras = {

state: {

transd: 'TRANS001',

workQueue: false,

services: 10,

code: '003'

}

};

this.router.navigate(['newComponent'], navigationExtras);

In newComponent

test: string;

constructor(private router: Router) {

const navigation = this.router.getCurrentNavigation();

const state = navigation.extras.state as {

transId: string,

workQueue: boolean,

services: number,

code: string

};

this.test = "Transaction Key:" + state.transId + "<br /> Configured:" + state.workQueue + "<br /> Services:" + state.services + "<br /> Code: " + state.code;

} //This how we can transfer securely without showing in path, Just remember the way no need of syntaxes

const routes: Routes = [

{path: 'admin', component: ProjectComponent,

children: [

{ path: 'list', component: ProjectListComponent},

{ path: 'create', component: ProjectCreateComponent } ] }

]

These are nested routes or child routes, we can call them like admin/list and admin/create etc. parent/child. If we want to create separate sub routes for same parent address we use children option.

Route Guard or Authentication

It is not ideal to allow navigate to any url, like you cant access cart link unless logged in. We use inbuilt angular’ss CanActivate interface for checking. This is a guard

ng g guard Access //access is name of guard, this will create follow and we implement inside CanActivtae(). This is a type of service so we declare inside providers.

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

import { CanActivate, Router } from '@angular/router';

import { UserService } from './user-service';//another service file with logic

@Injectable()

export class AccessGuardService implements CanActivate {

constructor(private serv:UserService, private router: Router ) {};

canActivate() {

let data = this.serv.canLoginToday()

if(data) {

this.router.navigate(['/login']) // if we return true then it will allow navigating to home path

} else {

return false;

} } }

We will tell to use this guard in routes

const routes:Routes = [

{path: ''redirectTo:"home", pathMatch: 'full',},

{path:"home",component:HomeComponent ,canActivate:[AccessGuardService]},

{path:"login",component:SuccessComponent}

];

Both nesting components and routing can somehow gives the same result which is loading the component but we have to use nested only if both the components are directly related like a product list > product detail. Cause every list has details to it. If relation is irrelevant and we want to navigate to something then we should use routing. Like my cart details from my dashboard. I can go anywhere from dashboard no relation.

## Jasmine

It’s a unit testing framework, so we will test each unit by unit. We write test cases in .spec.ts generated with component. So to test a service, pipe, component or anything first we need that environment for that we use **TestBed.** In code we create the environment using TestBed.configureTestingModule( env declarations).compileComponenets() .We give our testing component or whatever need inside declarations inside and ask it to compile them.

To run all the test cases mentioned in all spec.ts file we give: ng test

import { async, ComponentFixture, TestBed } from '@angular/core/testing';

import { WelcomeComponent } from ‘./welcome-component’; import rest

describe('WelcomeComponentTesting', () => {

let comp: WelcomeComponent;

let fixture: ComponentFixture< WelcomeComponent >;

beforeEach(async(() => {

TestBed.configureTestingModule({

declarations: [WelcomeComponent],

imports:[HttpClientModule,FormsModule,ReactiveFormsModule,RouterTestingModule],

}).compileComponents();

}) );

beforeEach(() => { //can also continue beforeEach function also

fixture = TestBed.createComponent(WelcomeComponent);

comp = fixture. componentInstance;

fixture.detectChanges();

});

it('should have title on the page', async(() => {

expect(comp.pageTitle).toEqual('Welcome');

}));

it('Should hide message', ()=> {

expect(comp.hide()).toBeFalsy();

});

it('Message',() =>{

let a1= fixture.nativeElement.querySelectorAll('div') //Can get all div elements like this

let value = a1[1].innerHTML.trim; //accessing 2 div element

expect(value).toEqual('Success is achived through accurate efforts'.trim)

}); //.trim will ignore blank or white spaces before and after

Since it is a component we need environment, for simple pipe we don’t need it.

import { AlphaCasePipe } from './alpha-case.pipe';

describe('AlphaCasePipe', () => {

let pipe;

beforeEach(() => {

pipe = new AlphaCasePipe();

});

it('create an instance', () => {

expect(pipe).toBeTruthy();

});

it('Transform to UpperCase',()=>{

let testInput = 'mY iNfY';

expect(pipe.transform(testInput,'U')).toMatch('MY INFY');

}); });

Some test cases mainly services need mocking( if we replicate what will come from backend that is mocking). Suppose there is a MessageService which has getName() which return observable from backend, then we create a mockService with same getName() separate file and we return our hardcoded data inside it (has to use *of*  keyword to convert into observable). Now in creating TestBed we say use mockService instead of MessageService as shown below. Also if routes are used in out component then we have to create mock routes also including mock base href

import {APP\_BASE\_HREF} from '@angular/common';

describe('AppComponent', () => {

const routes:Routes=[

{path:'message',component:MessageComponent}

]

beforeEach(async(() => {

TestBed.configureTestingModule({

declarations: [

AppComponent,

MessageComponent, //have to import,declare,provide everything required else test fails

HelloPipe

],

imports:[RouterModule.forRoot(routes)],

providers: [{provide: APP\_BASE\_HREF, useValue : '/' }, { provide:MessageService, useClass:MockService}]

}).compileComponents();

it('should be created', () => {

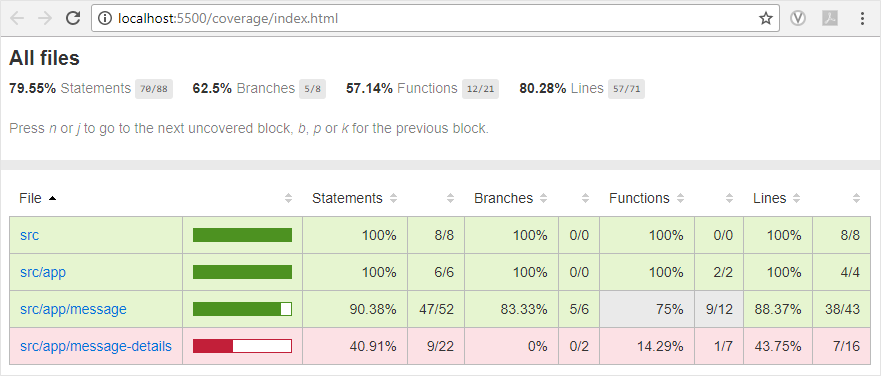
const service: MessageService = TestBed.get(MessageService); //not sure how this works,

expect(service).toBeTruthy(); no need of depth

});

We can see our code coverage like how much % code is covered in test cases by **Istanbul** tool by cmd ng test --code-coverage

This command will shows results in output terminal and also karma gets opened in browser showing how many test cases passed and failed and also create new coverage folder inside project which consists index.html shows code coverage report



## Styling

For default styling like google’s predefined themes etc. we can use Material Design. We will install it by these commands npm install @angular/material –save ; npm install @angular/cdk –save;

npm install @angular/animations –save;

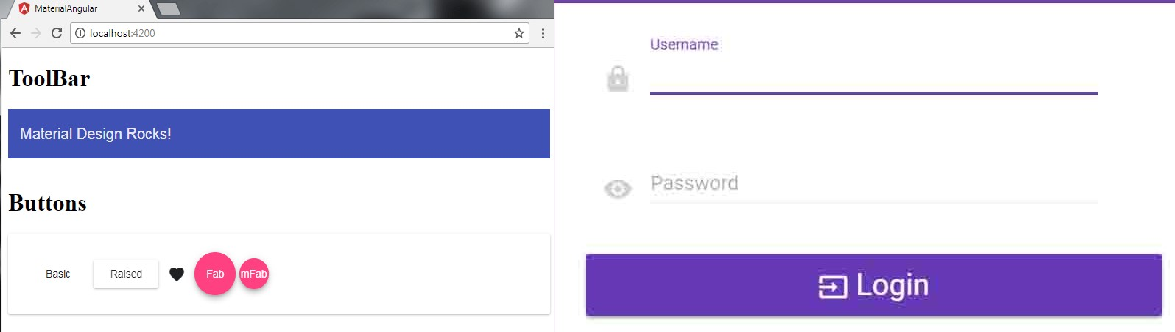
To use the themes in angular.json file we will paste this

"styles": [

"node\_modules/@angular/material/prebuilt-themes/indigo-pink.css"

],

Another theme: [node\_modules/@angular/material/prebuilt-themes/pink-bluegrey.css](mailto:node_modules/@angular/material/prebuilt-themes/pink-bluegrey.css)



To make button look stylish like this just import and declare those modules in imports of app.module

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

import {MatButtonModule} from '@angular/material/button';

imports: [

MatButtonModule, BrowserAnimationsModule],

providers: [],

Note: To open a dialog box we have to use MatDialog :

In html to open dialog box

<ol>

<li>

<mat-form-field>

<mat-label>What's your name?</mat-label>

<input matInput [(ngModel)]="name"> // we are taking this name to pass into dialog box

</mat-form-field>

</li>

<li>

<button mat-raised-button (click)="openDialog()">Pick one</button> //invokes dialog open fn

</li>

<li \*ngIf="animal">

You chose: <i>{{animal}}</i> // we will display this what we entered in dialog box

</li>

</ol>

In ts file:

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

import {MatDialog, MatDialogRef, MAT\_DIALOG\_DATA} from '@angular/material/dialog';

//MatDialog is dialogbox, ref is for reference, MAT\_DIALOG\_DATA is to transfer data

@component({‘selector’:’dialog-example’}……)

Export class….constructor(public dialog: MatDialog)….

openDialog(): void { //calling this function opens dialog box

const dialogRef = this.dialog.open(DialogOverviewExampleDialog, {

width: '250px', //setting up the dimensions

data: {name:this.name,animal:this.animal} //declaring variables what we use

});

dialogRef.afterClosed().subscribe(result => { //capturing result from dialog box

console.log('The dialog was closed');

this.animal = result;

});

} // we will create below new component in same ts file

@Component({

selector: 'dialog-overview-example-dialog',

templateUrl: 'dialog-overview-example-dialog.html',//In same folder

})

export class DialogOverviewExampleDialog {

constructor(

public dialogRef: MatDialogRef<DialogOverviewExampleDialog>,

@Inject(MAT\_DIALOG\_DATA) public data: DialogData) {}

// we are injecting the data we created in above openDialog()

onNoClick(): void {

this.dialogRef.close();

}

}

In html of 'dialog-overview-example-dialog.html in same comp folder

<h1 mat-dialog-title>Hi {{data.name}}</h1>

<div mat-dialog-content>

<p>What's your favorite animal?</p>

<mat-form-field>

<mat-label>Favorite Animal</mat-label>

<input matInput [(ngModel)]="data.animal">

</mat-form-field>

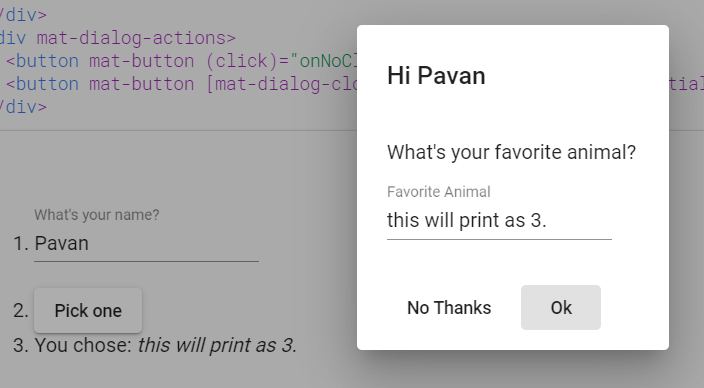
</div>

<div mat-dialog-actions>

<button mat-button (click)="onNoClick()">No Thanks</button>

<button mat-button [mat-dialog-close]="data.animal" cdkFocusInitial>Ok</button>

</div> //mat-dialog-close will invoke dialogRef.afterClosed() above and assigns animal



Here name given is sent to dialogbox an animal written is read and displayed.

<https://material.angular.io/components/dialog/examples>

**PrimeNG :** PrimeNG is a collection of rich UI components for Angular. It is an open source library. This library offers such components as Accordion, Panel, Grid, Dialog, Menu, charts, drag and drop, and many more. It also offers a variety of themes and templates

npm install primeng --save

npm install font-awesome –save or combine like npm I primeng font-awesome --save

import {ButtonModule} from 'primeng/button'; //declare needed inside imports of app.module.ts

"styles": [ //also has to declare inside angular.json

"node\_modules/font-awesome/css/font-awesome.min.css", // Makes use of font-awesome

"node\_modules/primeng/resources/themes/nova-light/theme.css", //Makes use of nova-light theme

"node\_modules/primeng/resources/primeng.min.css",

], //declare what we want to use in app.module.ts in imports

import {InputTextModule} from 'primeng/inputtext';

import {RadioButtonModule} from 'primeng/radiobutton';

import {CheckboxModule} from 'primeng/checkbox';

import {ButtonModule} from 'primeng/button';

import {RatingModule} from 'primeng/rating';

import {MessagesModule} from 'primeng/messages';

import {MessageModule} from 'primeng/message';

import {GrowlModule} from 'primeng/growl';

import {PanelModule} from 'primeng/panel';

No need write anything in component ts can use directly inside html

<p-radioButton name="rb" value="val1" label="English" ></p-radioButton>

<p-checkbox name="cb" value="val1" label="History" ></p-checkbox>

<p-rating [(ngModel)]="rating"></p-rating>

<button pButton type="button" label="Add book" (click)="addBook()"></button>

<p-messages [(value)]="msgs"></p-messages> //to display info messages

addBook() {

this.msgs = [];

this.msgs.push({severity:'info', summary:'Book Added', detail:' : New book Harry Potter added successfully with id 1001'});

} //this the format for primeng to display or understand

<p-chart type="doughnut" [data]="presalesData"></p-chart>

constructor() {

this.presalesData = { //this how chart data has to be

labels: ['January', 'February', 'March'],

datasets: [

{

label: 'Book presales',

data: [65, 59, 80],

backgroundColor: [

"#FF6384",

"#36A2EB",

"#FFCE56"

],

hoverBackgroundColor: [

"#FF6384",

"#36A2EB",

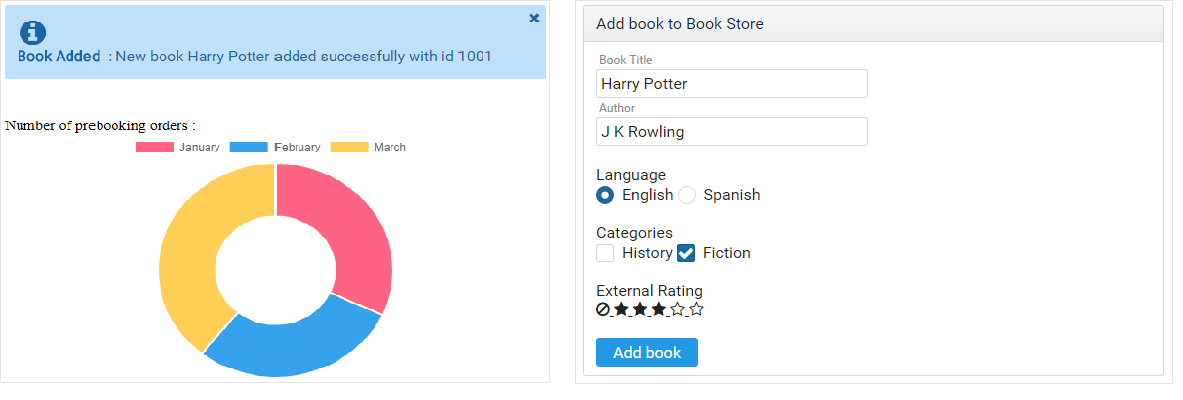
"#FFCE56"

]

}

]

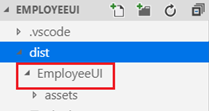
} }



## Build and Deploy

To deploy on cloud like Azure, we need to build

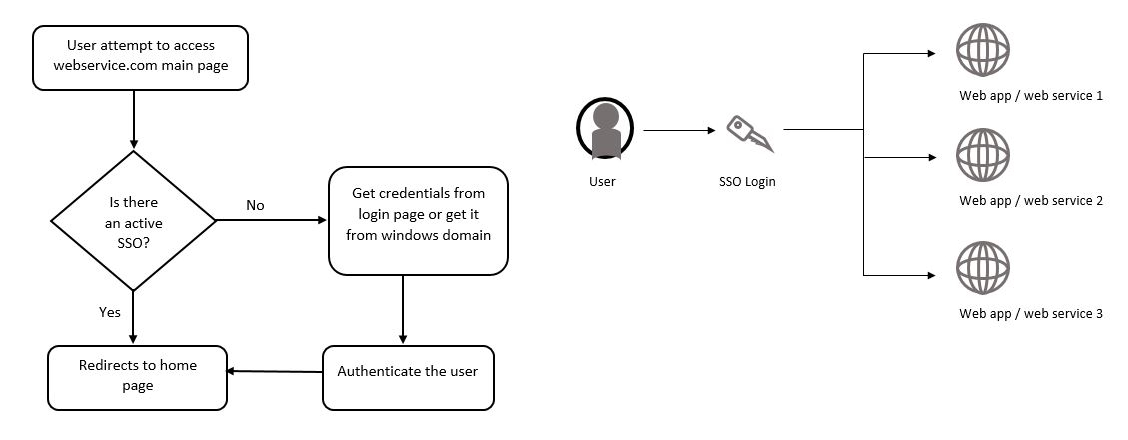
ng build ; this will create **dist**  folder in project which has to be deployed in cloud. Just like how we deployed express in azure same process will be done by selecting **Azure App service** in VS code, create a new webapp service and select Node LTS as environment and deploy. Select the project folder inside dist folder as shown below and browse the website on the hosted cloud link.



## Authentication

### Single Sign On (SSO)

If we login google gmail we can access google drive, yt etc. This is SSO.



If SSO available, no need multiple username passwords, no need to validate and login everytime, multiple logins is not possible. This is backend (node & express) thing, in front do we simply call the backend like getData() below in service

this.appservice.getData().subscribe(

(success) => { console.log(success, typeof success);

this.successmessage= success['username'];

console.log(this.successmessage, typeof this.successmessage);

},

(err) => {

this.errorMessage = err.error.message;}

getData() { //this is the service file

return this.http.get('http://backend-url/',{withCredentials:true}) }

In app.js of our express application, npm i node-sspi

var cors = require('cors')

var nodeSSPI = require('node-sspi')

var corsOptions = { // these are cors options not related to sspi

origin: ['http://frontend-server'],

optionsSuccessStatus: 200,

credentials:true

}

app.use("/",cors(corsOptions),function (req, res, next) { // don’t care about ‘/’,cors(options) here

var nodeSSPIObj = new nodeSSPI({ //creating an sspi object

retrieveGroups: true })

nodeSSPIObj.authenticate(req, res, function(err){ //method to authenticate the logged in user

res.finished || next() //.authenticate() will return error if not authenticated which makes response

}) }) finished or goes to next handler, if successful then executes below app.use

app.use(cors(corsOptions),function(req, res, next) { cors are optional here, directly can write funcs

userName = req.connection.user // if authentication success then req will be filled with details

if(userName){ which we can access like this

uName = userName.substr(11,) // to display only the user name

var out = {}

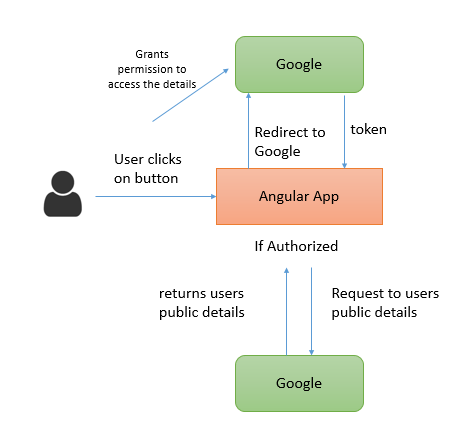
out["username"] = uName; } //this will be captured in ts file of frontend above as success[“usernam

res.send(JSON.stringify(out))

})

### OAuth

Open Authorization(OAuth) .To use credentials of Google,Facebook to login to our application we use OAuth of angular. User clicks on login button, redirects to google and login there, google validate user and generates token, we can use that to access users public profile data. Follow below steps



Bjm n

1)Create a Google account 2) Register our App in Google cloud Platform API credentials and generate Client token by registering our project.

3)Install an OAuth library in Angular app, 4)Use the client token in the Angular app

5)Invoke the libraries API to contact Google and handle the response.

npm install angularx-social-login –save

import { SocialLoginModule } from **'angularx-social-login'**; //app.module.ts

import { AuthServiceConfig, GoogleLoginProvider } from **'angularx-social-login'**;

const config = new AuthServiceConfig([ //app.module.ts

{

id: GoogleLoginProvider.PROVIDER\_ID,

provider: new GoogleLoginProvider('<Your client ID token generated by Google>')

},

imports: [

BrowserModule, FormsModule, HttpModule, SocialLoginModule

],

providers: [

{

provide: AuthServiceConfig,

useFactory: config

}

]

No need of service file, angular provides default service for this type. In component ts file

import { AuthService, SocialUser, GoogleLoginProvider } from 'angularx-social-login';

export class DemoComponent implements OnInit {

user: SocialUser;

constructor(private authService: AuthService) { }

ngOnInit() {

this.authService.authState.subscribe((user) => { this.user = user; console.log(user); }); }

signInWithGoogle(): void { //Invoke this fn on click of login with google

this.authService.signIn(GoogleLoginProvider.PROVIDER\_ID).then(x => console.log(x));

}

signOut(): void { //these signIn,signOut are default functions

this.authService.signOut();

}

}

oAuth is which angular uses to login by 3rd party, Auth0 is 3rd party login provider outside

## Testing Service & Routing

We write test cases inside .spec.ts file. For example Product service

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

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

import { Observable } from 'rxjs/Observable';

import { Message } from './message'; // it is a class we defined just give type(not required)

@Injectable()

export class ProductService {

constructor(private http: HttpClient) { }

url = '/assets/message.json';

getData(): Observable<Message[]> {

return this.http.get<Message[]>(this.url);

}

}

In product.service.spec.ts (HttpTestingController makes dummy requests and response)

import { async, TestBed, inject } from '@angular/core/testing';

import { HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';

import { Observable } from 'rxjs/Observable';

import { ProductService } from './product.service';

import { Product } from './product'; //same like message just to define structure

describe('ApiService', () => {

let httpMock: HttpTestingController; // Has to create httpMock to mock our request

let productService: ProductService; //Making our product service instance

beforeEach(async(() => {

TestBed.configureTestingModule({ //configuring test bed

// Providing HttpTesting module instead of original HttpClientModule

imports: [HttpClientTestingModule],

providers: [

ProductService

]

}).compileComponents();

// Getting the reference of all mocked services

httpMock = TestBed.get(HttpTestingController); //after compiling we are using the testbed

productService = TestBed.get(ProductService);

}));

// Checking service is injected successfully and instance is created or not

it('should be created', inject([ProductService], (service: ProductService) => {

expect(service instanceof ProductService).toBe(true);

}));

// Checking for getProducts Method

describe('invoking getProducts method', () => {

// Creating variables to store results

let result;

let response: Product[];

let request;

beforeEach(() => {

// Assigning dummy value

productService.producttype = 'tablet';

// Creating mock response value

response = <Product[]>[];

// Invoking function

productService.getProducts() .subscribe(data => {

result = data; // storing the mocked response under result

}); // this is the original way, we are invoking this to get data to compare later

// Expecting request under particular url

request = httpMock.expectOne('assets/products/tablets.json');

// Flushing the mocked response means storing the response from the above mock request

request.flush(response); inside the variable response

// Ensuring there is no standing calls means service is completed without any call backs

httpMock.verify();

});

// Checking the request is made under GET method

it('should make request under GET method', () => {

expect(request.request.method).toEqual('GET');

});

// Checking the result is populated with mocked value, both original way a mocking gives same

it('should return value from backend', () => {

expect(result).toEqual(response);

});

});

});

Since we know that getProducts() in service uses that url and gives the data we created a mock as it is and compared here. Sometimes if we know the response then we can directly use that response to compare instead of calling original service for the result.

it('should return observable',inject([HttpTestingController,DatareaderService],(httpMock,service ) => {

const mockResponse = '{"id":"1","message":"You can do anything, but not everything.","by":"David Allen"}';

service.getData().subscribe( (response) => {

expect(response).toBe(mockResponse)

}) Instead of mocking, we directly tested service by compared with string we given.

Finally, there are 2 ways of testing a service. 1) Call original service and store the response and then create a mock of that service like hitting the same url which we hit inside service and compare the both responses like first example 2) Call original service and compare with the string like above example without mocking. In this case if we want to create an observable use **of** keyword.

**Routing Testing**: For routing testing we use below which are provided by angular

RouterTestingModule : which helps in setting up the router, which actually doesn't change the URL.

fakeAsync : since routing is an asynchronous activity, we use fakeAsync which is an asynchronous testing method.

navigate : is used to trigger the router, to navigate to a specific path.

tick : is used to wait, until all the asynchronous operations are complete.

location : is used to inspect the path of our application.

import {Router,Routes} from'@angular/router';

import { MessageDetailsComponent } from './message-details/message-details.component';

import { TestBed, fakeAsync, tick } from '@angular/core/testing';

import { RouterTestingModule } from '@angular/router/testing';

import {Location} from '@angular/common'

import { AlphaCasePipe } from './message/alpha-case.pipe';//use pipes if declared any in component

describe('Router: App', () => {

let location: Location;

let router: Router;

let fixture;

let routes = [

{path: 'messageDetails/:id', component:MessageDetailsComponent}

]

beforeEach(() => {

TestBed.configureTestingModule({

imports: [RouterTestingModule.withRoutes(routes)], //original RouterModule.forRoot(routes)

declarations: [

MessageDetailsComponent ,

AlphaCasePipe

],

providers : [Location]

});

router = TestBed.get(Router);

location = TestBed.get(Location);

});

it('navigate to /messageDetails/:id', fakeAsync(() => {

router.navigate(['/messageDetails/1']);

tick();

expect(location.path()).toBe('/messageDetails/1');

}));

});

## Build

All the components, pipes,services, html everything has to be rendered as JS (from ts transpiling) to run it in the browser, this is done by build. They are 2 types

**JIT (Just In Time)**: Compiles angular app during run time in browser (default method, ng build followed by ng serve) If not mentioned then it take as JIT

**Ahead-Of-Time(AOT):**  compiles Angular app at build time (not default has to mention the type of build, ng build –aot followed by ng serve --aot)

AOT is preferred because

* **Faster rendering** by browser because it’s a pre-compiled application
* **Smaller Framework Download Size:** As the app is pre-compiled, there is no need of downloading Angular compiler in the browser which is roughly half of the size of Angular itself. This dramatically reduces the application payload.
* **Detects Template Errors Early:**The AOT compiler detects the template binding errors during the build setup itself.
* **Better Security:**AOT compiles components and HTML templates before it is served to the client. With no templates to read and no risky client side HTML or Javascript evaluation, there are less chances for injection attacks.
* **Fewer Asynchronous Requests:** AOT compilation inlines external HTML templates and CSS stylesheets into the application itself, eliminating separate ajax requests for those files.

To generate an optimized build we use production type build (ng build –prod)

This will enable AOT compiler, enables production mode, concatenates many files into bundles, extra spaces & comments removed, uglifies the app by rewriting the variables and function names into smaller ones, removes the unused code and modules called dead code elimination. And finally build process creates dist folder with above rules and we use that for deployment

## JWT (JSON Web Token)

JSON Web Token (JWT) is an open standard ([RFC 7519](https://tools.ietf.org/html/rfc7519)) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed.

These are mainly used to

1) **Authentication**: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.

2) **Information Exchange**: JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

In its compact form, JSON Web Tokens consist of three parts separated by dots (.), :

* 1. Header
  2. Payload
  3. Signature eg: xxxx.yyyyy.zzzzz (checkout below image)

The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: registered, public, and private claims.

[**Registered claims**](https://tools.ietf.org/html/rfc7519#section-4.1): These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: **iss** (issuer), **exp** (expiration time), **sub** (subject), **aud** (audience), and [others](https://tools.ietf.org/html/rfc7519#section-4.1).

[**Public claims**](https://tools.ietf.org/html/rfc7519#section-4.2): These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the [IANA JSON Web Token Registry](https://www.iana.org/assignments/jwt/jwt.xhtml) or be defined as a URI that contains a collision resistant namespace.

[**Private claims**](https://tools.ietf.org/html/rfc7519#section-4.3): These are the custom claims created to share information between parties that agree on using them and are neither registered or public claims.

Signature: To create the signature part you have to take the encoded header, the encoded payload, a secret

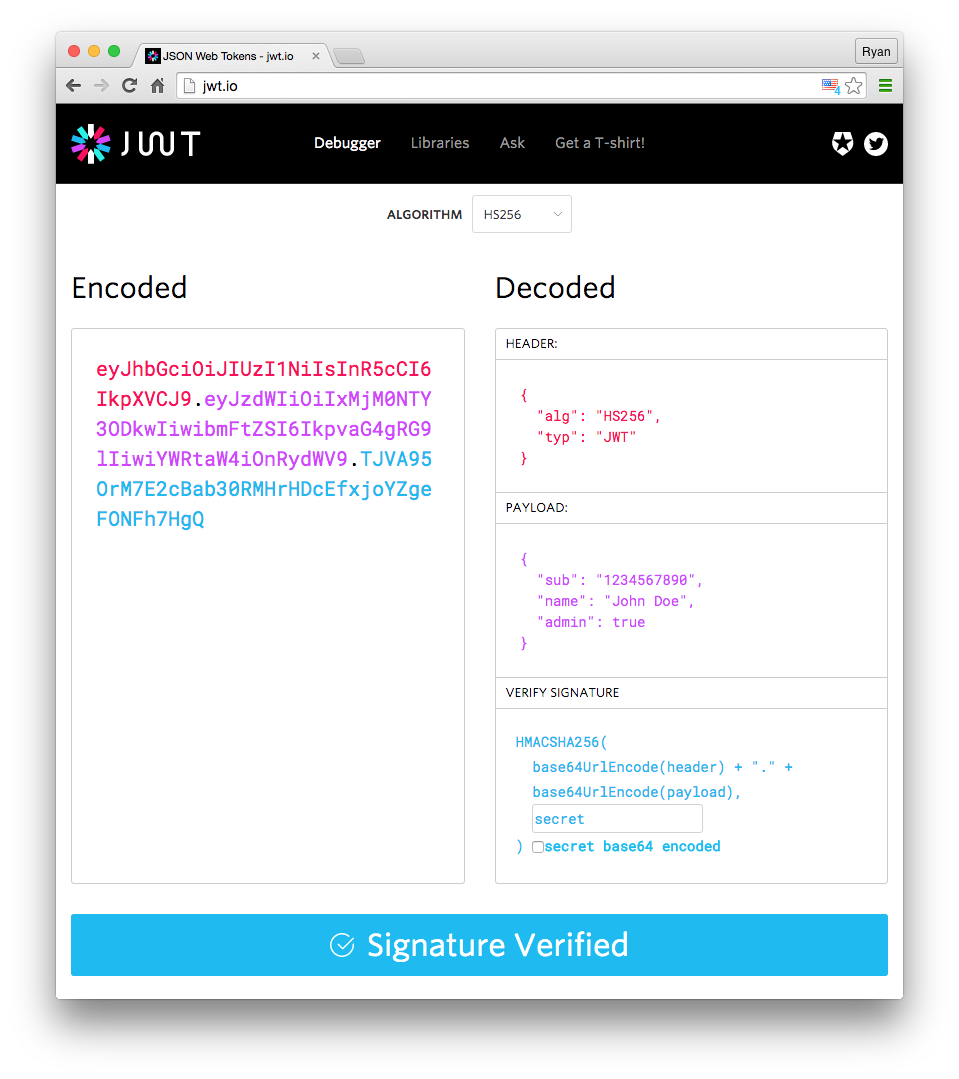
HMACSHA256(

base64UrlEncode(header) +’.’+

base64UrlEncode(payload),

secret)

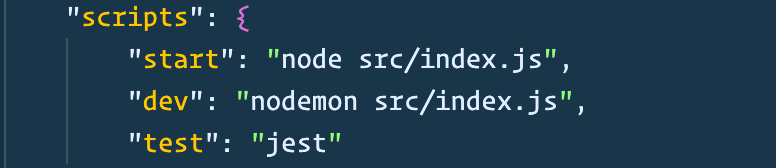
This creates following encoded message which will be passed to the backend in the Http request header under Authorization Bearer <encoded-message>.



So when user logins in, JWT tokens are generated and when user makes API request we send these tokens with it which will be verified by backend to process it or not

# Jest

To test our API in backend (express) we can use Jest. npm i jest, test files should be under \_\_test\_\_ folder inside src folder of app and should end with .spec.js each. Inside this we write test cases just like how we write jasmine test cases in Angular using describe and it. We run test cases by simple command jest or we define in package.json and use that like npm run test





## NgRX – Redux

Used to store the current state and actions which are processed by reducer.

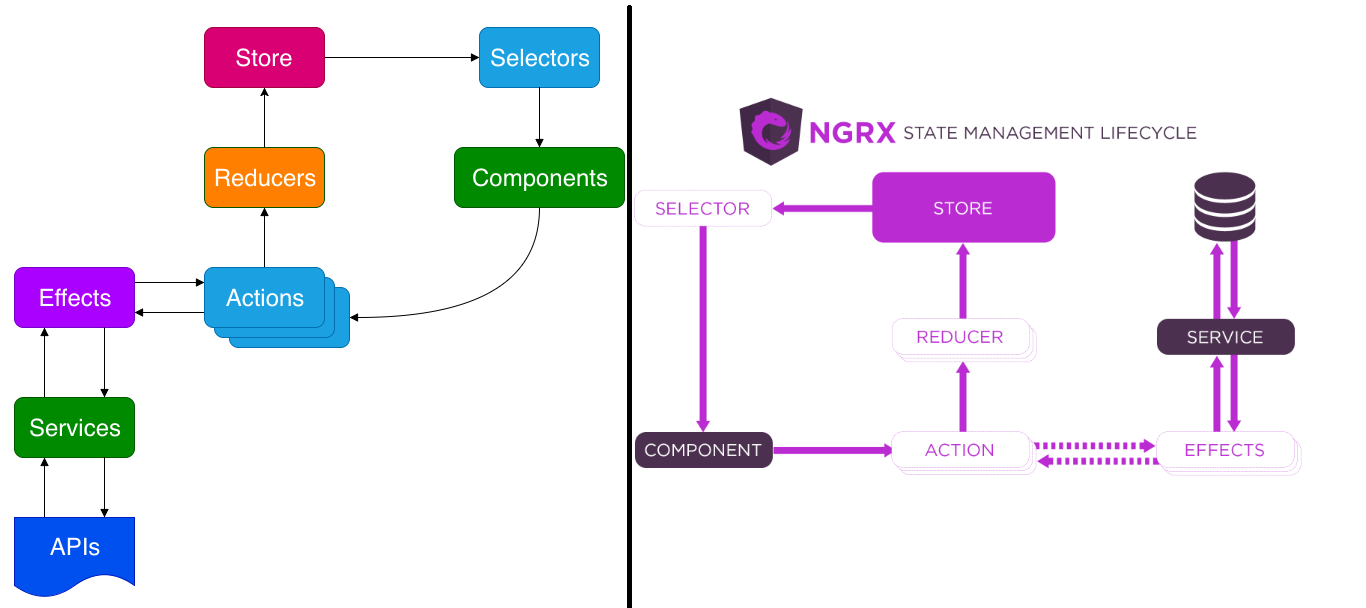
<https://medium.com/codingthesmartway-com-blog/angular-and-redux-ecd22ea53492>

Mainly NgRX store is used for state management of our application, it is a Redux implementation of Angular, npm i @ngrx/store followed by ng add @ngrx/store

## NgRx App State Management

Let’s look at the NgRx implementation — there are several components to understand.

* **Store**: Store is what holds the app's state.
* **Action**: A unique event dispatched from components and services that describe how the state should be changed. For example, ‘Add Customer’ can be an action that will change the state (i.e. add a new customer to the list).
* **Reducer**: All the state changes happen inside the reducer; it responds to the action and, based on that action, it will create a new immutable state and return it to the store.
* **Selector**: Selector is a function used for obtaining a part of the state from the store.
* **Effect**: A mechanism that listens for dispatched actions in an observable stream, processes the server response, and returns new actions either immediately or asynchronously to the reducer to change the state. Please note that we are not using effect in this example app.



1. Your application’s **state** is maintained in the **store**. The **store** is immutable.
2. Your application’s **components** can subscribe to the **store** and get automatic updates of **state** through **selectors**.
3. **Selectors** enable **components** to get a **slice** (a part) of your application’s **state**, and also mutate state with selector functions.
4. **Actions** modify the state of the **store** by using **reducers** (functions) that enable changes while keeping it immutable.
5. **Meta-Reducers (not shown)**are hooks where you can pre or post-process actions before they get invoked.
6. **Effects** occur as a result from **actions** and can also create **actions** when called. **Effects** primary responsibility is to create async side-effects (like service calls to APIs), that ultimately generate other **actions**.

### Hands On

Create a basic class customer with name property and export it, to be used as annotation. Create action.ts file and create 2 actions AddCustomer and RemoveCustomer as shown below, created ActionEx just to make our action classes implement payload also, cause Action class implements only type property, we can also skip enum instead we can give direct values, but this is for maintainability. Create Reducer.ts which changes the state according to the action. As you can see, we just made switch case according to predefined actions like if action is ‘[Customer Component] Add’ in parameter then reducer will add the given payload to the state list and returns it. Same for remove then it is removing the given customer from state list by doing slice operations. If not matches, then state is returned as it is. So, reducers are pure functions, they always take state and alter it and returns it without changing its properties, both input and output is same.



Now to make this reducer available throughout the application we need to add this in app.module.ts, However it is dependent on and alter store which is global so we will mention like

import { StoreModule } from '@ngrx/store';   
import { CustomerReducer } from './customer.reducer';

@NgModule({

imports: [

BrowserModule,

StoreModule.forRoot({ customers: CustomerReducer })

]

In componenet.ts to access the state and to invoke actions

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

import {Customer} from '../models/customer';

import {Observable} from 'rxjs';

import {select, Store} from '@ngrx/store';

@Component({

selector: 'app-customers-view',

templateUrl: './customers-view.component.html',

styleUrls: ['./customers-view.component.css']

})

export class CustomersViewComponent {

customerList: Observable<Customer[]>;

constructor(private store: Store<{ customers: Customer[] }>) {

this.customers = store.pipe(select('customers'));

}

}

In html, we display all in list

<h4>List of Customers</h4>

<ul class="customers">

<li \*ngFor="let customer of customers | async; let i=index">

<span >{{i+1}}.</span> {{customer.name}}

</li>

</ul>

Now to add a customer, create a customerAdd componenet and give below code. Use with selector

<h4>Add New Customer</h4>

<input #box ><button (click)="AddCustomer(box.value)">Add</button>

Inside component.ts

AddCustomer(customerName: string) {

const customer = new Customer();

customer.name = customerName;

this.store.dispatch(new CustomerAdd(customer));

}

for remove, in the customers view, add a button and send the index

<h4>List of Customers</h4>

<ul class="customers">

<li \*ngFor="let customer of customers | async; let i=index">

<span >{{i+1}}.</span> {{customer.name}}

<button style="float: right" (click)="removeCustomer(i)">Remove</button>

</li>

</ul>

Inside componenet.ts

removeCustomer(customerIndex) {

this.store.dispatch(new CustomerRemove(customerIndex));

}

We don’t anywhere in the code called customerReducer function, we created actions and we called that action in our component.js according to UI. Due to we mentioned CustomerReducer in app.module.ts for store module ‘customers’ these actions are automatically handled by this reducer. In component.ts also we initiated store.pipe select customers, this is how everything is managed.

Instead of creating action class and assigning type etc. and reducer explicitly access action.type and payload we can use directly default fns (this is from Version 8 of ngRx & Angular)

import { createAction } from '@ngrx/store';

// createAction directly creates Action with type as given

export const increment = createAction('[Counter Component] Increment');

export const decrement = createAction('[Counter Component] Decrement');

export const reset = createAction('[Counter Component] Reset');

In reducer.ts

import { createReducer, on } from '@ngrx/store';

import {increment, decrement, reset } from './counter.actions';

export const initialState = 0; //createReducer will create reducer

const \_counterReducer = createReducer(initialState,

on(increment, state => state + 1),

on(decrement, state => state - 1),

on(reset, state => 0),

);

export function counterReducer(state, action) {

return \_counterReducer(state, action);

}

ng generate reducer store/reducers/auth --reducers index.ts (cmd to create reducer with skeleton)

### Effects

🡪Generally, components call services and interact with outside Angular services, instead effects provide way to interact and make components don’t need to worry about services. Effects isolate side effects from components, allowing for more pure components that select state and dispatch actions.

🡪Effects are long-running services that listen to an observable of every action dispatched from the [Store](https://ngrx.io/guide/store).

🡪Effects filter those actions based on the type of action they are interested in. This is done by using an operator.

🡪Effects perform tasks, which are synchronous or asynchronous and return a new action.

**Hands On:** Now consider basic old service-based approach like below, In component.ts

constructor (private movieService: MoviesService) {}

ngOnInit() {

this.movieService.getAll().subscribe(movies => this.movies = movies);

}

In movieService.ts we have a http get method

export class MoviesService {

constructor (private http: HttpClient) {}

getAll() {

return this.http.get('/movies');

}

}

As we can see above, component had to manage state, interact with service and again had to update store once the values are updated. In larger application this will make things complicated, use effect Effects when used along components will decrease responsibility of component and maintain easily So let’s implement effects into this service based app, in component.ts now dispatch action

export class MoviesPageComponent {

movies$: Observable<Movie[]> = this.store.select(state => state.movies);

constructor(private store: Store<{ movies: Movie[] }>) {}

ngOnInit() {

this.store.dispatch({ type: '[Movies Page] Load Movies'});//instead of service, we dispatched action

}

}

ng generate effect store/effects/movie --module app.module --root true (to generate effect file and also updates app.module.ts registers our effect file)

import { EffectsModule } from '@ngrx/effects';

import { MovieEffects } from './effects/movie.effects';

@NgModule({

imports: [

EffectsModule.forRoot([MovieEffects])

],

})

export class AppModule {}

In MovieEffect.ts (use createEffect() only if ngRx and Angular version 8 and above)

loadMovies$ = createEffect(() => this.actions$.pipe(

ofType('[Movies Page] Load Movies'),

mergeMap(() => this.moviesService.getAll()

.pipe(

map(movies =>new actions.add ({ type: '[Movies API] Movies Loaded Success', payload: movies })),

catchError(() => of({ type: '[Movies API] Movies Loaded Error' }))

))

)

);

For Angular version 7 and below

@Injectable()

export class AuthEffects {

constructor(private actions$: Actions,

private http: HttpClient) {}

@Effect()

loadMovies$: Observable<Action> = this.actions$.pipe(

ofType('[Movies Page] Load Movies'),

switchMap(() => {

return this.moviesService.getAll()

.pipe(

map((movies) => {

return new movieActions.add({ type: '[Movies API] Movies Loaded Success', payload: movies}),

catchError(() => of({ type: '[Movies API] Movies Loaded Error' })) //incase service errors

})

)

})

);

}

### RxJS built in functions

In rxjs there are several standalone functions (which are independent and not belongs to any class etc. so global like map, filter, scan…), these functions take observable and manipulates it and gives new observable and all these are chained with help of pipe function

**scan** – this function takes previous value and does operation and again takes this result and does operation with next value, Eg: basic value 2 is and first value 1 of observable is passed into scan fn which results 3 and prints it in subscriber and this 3 and next value 2 is passed into scan, result 5

import { of } from 'rxjs';

import { scan, map, filter } from 'rxjs/operators';

const source = of(1, 2, 3);

// basic scan example, sum over time starting with zero

const example = source.pipe(scan((acc, curr) => acc + curr, 2));

// log accumulated values

// output: 3,5,8

const subscribe = example.subscribe(val => console.log(val));

**map, filter –** These are similar fns we saw in arrays but these are imported and used for rxjs observables. So if we want to filter observable based on a condition then use this.

**Tap**: Can perform side effects with observed data but **does not modify** the stream in any way. Formerly called **do()**. You can think of it as if observable was an array over time, then tap() would be an equivalent to Array.forEach().

**mergeMap -** Projects each source value to an Observable which is merged in the output Observable.

Eg: const letters = of('a', 'b', 'c');

const result = letters.pipe( //i means number 0 to infinity so every 1sec a0,b0,c0,a1,b1,c1,a2….

mergeMap(x => interval(1000).pipe(map(i => x+i))),//taking letters observable and generates new

);

result.subscribe(x => console.log(x)); //subscribing to new result observer not old letters

**shareReplay –** it is used to prevent multiple triggers of the API call by accident, below post is triggered once

login(email:string, password:string ) {

return this.http.post<User>('/api/login', {email, password}).shareReplay();

# JWT User sessions

<https://blog.angular-university.io/angular-jwt-authentication/>

In the above JWT discussion, we saw how JWT tokens are look like and how they are encrypted. Now we will see in real time MEAN application. JWT is *A bearer token is a signed temporary replacement for the username/password combination!*

Authentication starts with a Login page, which can be hosted either in our domain or in a third-party domain. In an enterprise scenario (like infosys), the login page is often hosted on a separate server, which is part of a company-wide Single Sign-On solution (we login in microsoft portal to login infy). This third party can be Auth0 which authenticates user either by user ID/password or google, fb logins and then sends back the user identity in a token which is handled by OAuth2.0 in our application. By this way we don’t have to manage authorization code, techniques, Single Sign On can be achieved as same token is used across multiple apps, Multi Factor Authorization by OTP, RSA. By this method our application won’t handle credentials and authorization code, so a great method.

But still logging via our login page is still commonly used so lets use JWT for authorization. Let’s assume there is a login service in our component.ts which gets invoked when login button click

this.authService.login(val.email, val.password)

.subscribe(() => {

console.log("User is logged in");

this.router.navigateByUrl('/');

});

login(email:string, password:string ) { //in autheService file we have this login service function

return this.http.post<User>('/api/login', {email, password})

// this is just the HTTP call,

// we still need to handle the reception of the token

.shareReplay(); //to make sure post request is not called multiple times

}

Now validating user and generating JWT is backend work so lets create in express application

import \* as jwt from 'jsonwebtoken'; //all imports are done including jwt

app.route('/api/login').post(loginRoute); //route to loginRoute function

const RSA\_PRIVATE\_KEY = fs.readFileSync('./demos/private.key');//private key is signature

(private key is used to sign the token, public key is to verify them)

export function loginRoute(req: Request, res: Response) {

const email = req.body.email,

const password = req.body.password;

if (validateEmailAndPassword()) {

const userId = findUserIdForEmail(email); //gets userId for this email

const jwtBearerToken = jwt.sign({}, RSA\_PRIVATE\_KEY, {

algorithm: 'RS256', //encryption

expiresIn: 120,

subject: userId //payload

}

// send the JWT back to the user

// TODO - multiple options available

}

else {

// send status 401 Unauthorized

res.sendStatus(401);

}

}

Here private key is only stored in authentication server, public key to validate token can be hosted in a URL publicly and we can rotate that easily <https://angularuniv-security-course.auth0.com/.well-known/jwks.json>

Now to send back JWT token to client, we can use either Cookie, HTTP response header or body. Body has the response data so it’s generally not used, so mainly sent by cookie and header.

For Cookie we send like res.cookie("SESSIONID", jwtBearerToken, {httpOnly:true, secure:true});

Once the jwtBearerToken is generated in above process we are sending in cookie as SESSIONID, we have to set HttpOnly : true and secure: true this is for because cookies are automatically sent to every request client made, we don’t have to explicitly send JWT token for every request but if we click on any malicious sites then cookies are also sent to that site where hackers get valid JWT token which they can use to do operations on behalf of use (this is Cross Site Request Forgery - XSRF)

To avoid sending cookie to sites other than out of domain we kept httpOnly flag true, secure true allows us to secure the cookie by transmitting cookies using https protocol. By using this method as we discussed before in enterprise scenario the login page is hosted in separate server so our cookies won’t work, to avoid this third party authentication providers might allow use to run externally hosted login page in configurable sub domain of our website. Like our app is in app.example.com then third party authentication like Auth0 provides login in our domain like login.example.com Now cookies will be send since domain matches and gets assigned with Token and token is handled by oAuth | backend. If third party won’t allow login in our domain then fails.

Now let’s see how to send the token without cookies, we can send in response body

res.status(200).json({

idToken: jwtBearerToken,

expiresIn: ...

}); //By this approach there is no XSRF problem but we now need to handle token and write extra code to send token every http request we send to backend, also hackers can access token if response intercepted.

To handle token in our client app we use localStorage (this storage is same if we open multiple tabs), see the below authLogin service which sends response from http post method to setSession which stores userId and expiration time in localStorage variables. Loggedout will remove storage variables and isLogged in will get the expiration time within or not.



To send the token every time we request backend, we use **HttpInterceptor** which intercepts our req and assign JWT token in header Authorization field which is validated in backend by **expressJwt** using public key, if authorized then only middleware is called as shown below.



Here in validating token, we used public key from application server instead it's much better to have the Authentication server publish the JWT-validating public key in a publicly accessible Url. This gives advantage of simplified key rotation. If we want to publish new key we can publish in authentication server. Typically during periodic key rotation, we will have the two keys published and active for a period of time larger than the session duration, in order not to interrupt user experience, while a revocation might be effective much faster. There is no danger that the attacker could leverage the public key. The only thing that an attacker can do with the public key is to validate signatures of existing JWTs, which is of no use for the attacker.

To publish the keys we use JWKS (JSON Web Key Set). So we publish our public key in this and this gives output of below, kid is key identifier and x5c is public key. This key hosted in auth server.



To validate our JWT using this public key we use jwks-rsa, since the published key output format is same always the jwks-rsa will read it and extract key. URI will give public key on auth server (above).



Cache is for not fetching public key every time, cache period is 10hours, rateLimit is to limit number of time fetch publickey is done around 10time limit.

We use  **x-www-form-url-encoded** true in header whenever we make post request so that to not show the body of req.

We setup token in header x-access-token whenever we send request to backend & backend fetches

var token = req.headers['x-access-token'];

we will use jwt.sign() to sign and jwt.verify() to verify token or expressJWT to verify token

# OAuth vs Auth0

[**OAuth 2.0**](https://oauth.net/2/) is a protocol that allows a user to grant limited access to their resources on one site, to another site, without having to expose their credentials. Like we login using google, fb accounts in another sites. Standard protocol to implement authorization

[Auth0](https://auth0.com/) is a company that sells an identity management platform with authentication and authorization services that implements the OAuth2 protocol (among others). This is the one which validates user and sends a token back to the app which is handled by OAuth

# DOM in angular

To use DOM we can either use elementRef or @ViewChild

<input #name> //#name will bind template variable so we can use in component, id=”name” is different, it will not bind to any so it won’t work, always give # and id value same to solve this

@ViewChild ('name',{static:true}) public Name:ElementRef;

this.Name.nativeElement.value = "Default value";

Another method is injecting Elementref inside constructor and accessing by template variable

constructor(public el:ElementRef){}

this.el.nativeElement.querySelector('#usr').value = "working"

For events like click etc. use @ViewChild method, elementref will not work

<button #btn></button>

@ViewChild ('btn',{static:true}) button :ElementRef;

this.button.nativeElement.addEventListener('click',(event)=>{console.log(event)})

this will invoke when we click on button, we can also manually click this in code like

this.button.nativeElement.click();

{static:true} is given to make the @ViewChild elementref available in ngOnInit(){}, generally if false given then it is available only after ngViewInint(){}

In \*ngFor we give single #id and multiple div gets generated in that case we use @ViewChildren

To listen to every action like scroll, typing, focus etc. we use @HostListener, gets invoked everytime

@HostListener('keydown',['$event']) onKeyDown(event){

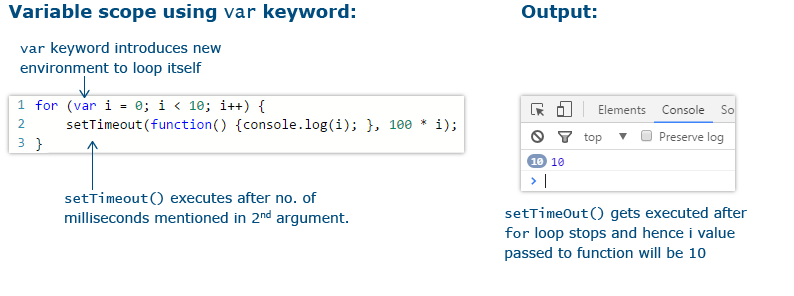
    let e = <KeyboardEvent>event;

    console.log(e.keyCode);

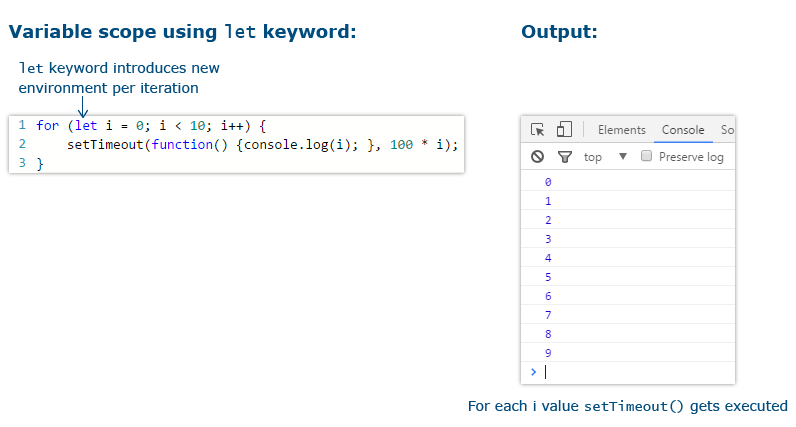
  }

# Declaring variables

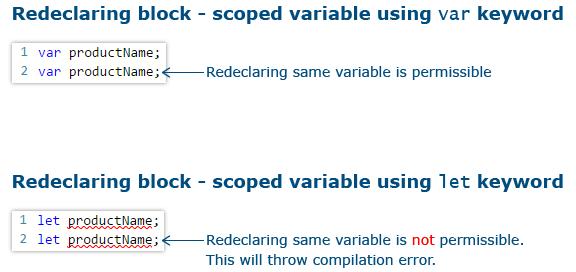
|  |  |
| --- | --- |
| **Data Type** | **Explanation** |
| var | * Variable declared with this type would have function scope. * They can be re-assigned and re-defined. * When Variable declared outside the function, it would have global scope and automatically attaches itself to the window object. |
| let | * Variable declared with this type would have a block-level scope. * They can be re-assigned and cannot be redefined. |
| const | * Variable declared with this type would have a block-level scope. * They can be neither re-assigned nor re-defined. |



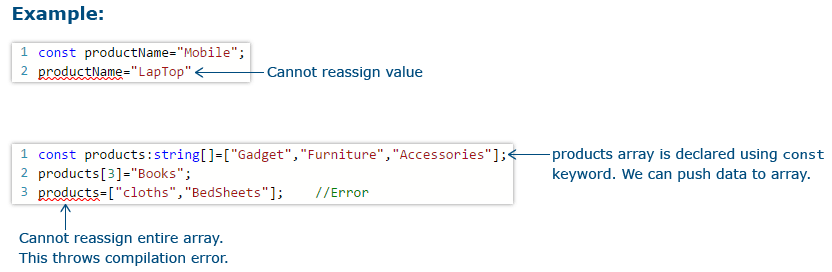
This is because var is functional scope but not block scope so by the time setTimeout reaches timer the loop is completed, and it will print the final 10 value 10 times because of loop



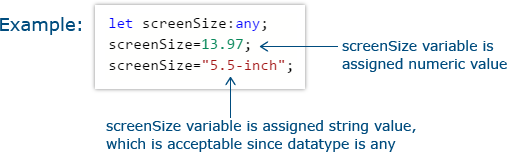
Here let is block scope so it is binded to each loop and the value is unique for every time loop runs so even after timer runs that i value is not incremented.



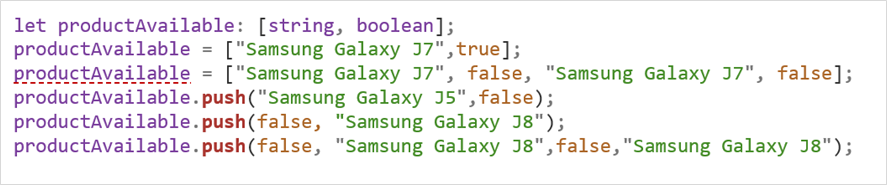
Const keywords are also block scope and enter value can’t be reinitialized or reassigned.



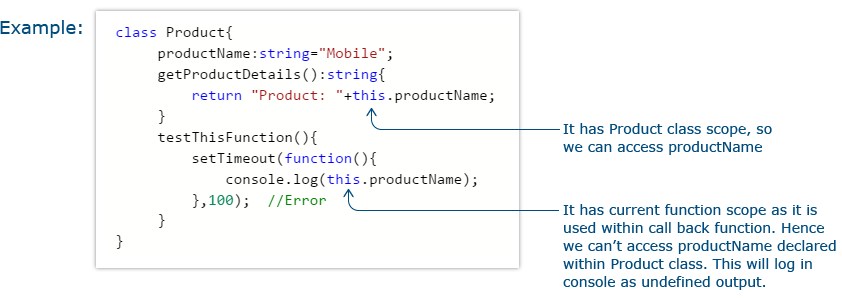
Type any means dynamical typing, type gets assigned based on value, if we don’t mention type it takes default any as type

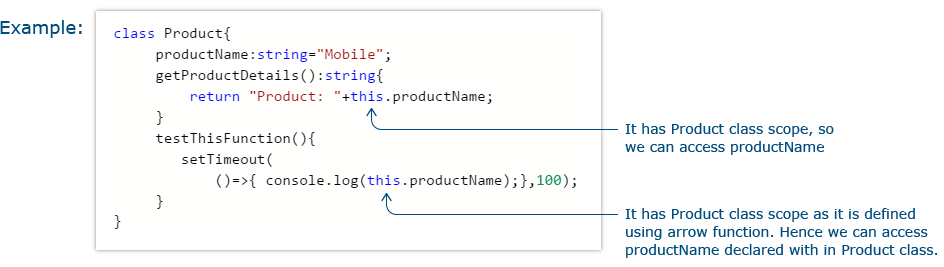


For tuple we can’t assign values directly as Typescripts restriction length policy, instead we need to push into tuple array. Tuples take different datatypes values unlike array which takes only one datatype like array1:string[] takes all string values only.

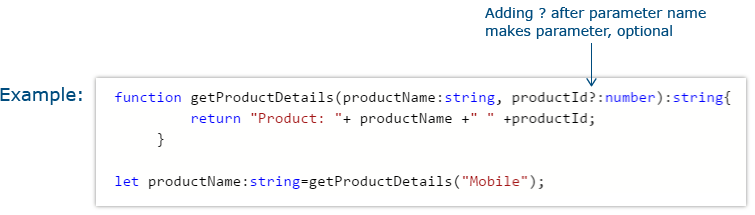


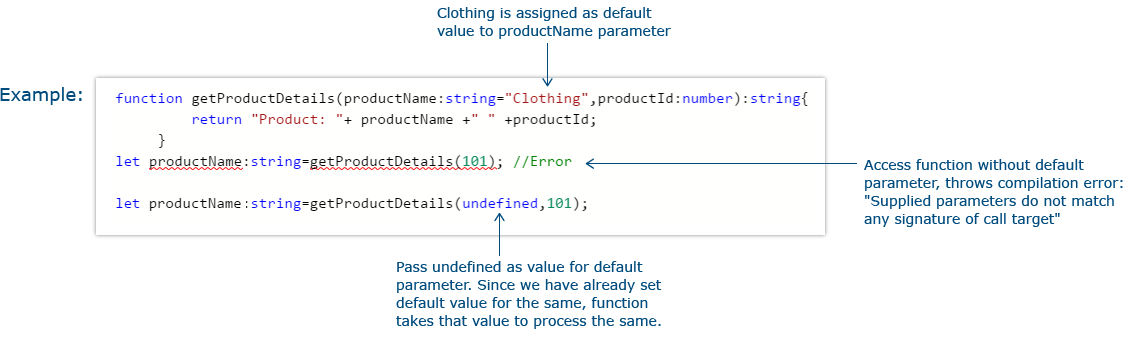
Using this and arrow functions





Optional and default parameters, all optional has to be at last





Duck typing means giving more variables than mentioned in interface

interface Product {

productId: number; //to make any property optional give? Eg: productid?:number

productName: string;

}

function getProductDetails(productobj: Product): string {

return 'The product name is : ' + productobj.productName;

} //below object declared new productCategory along with Prodcut interface properties

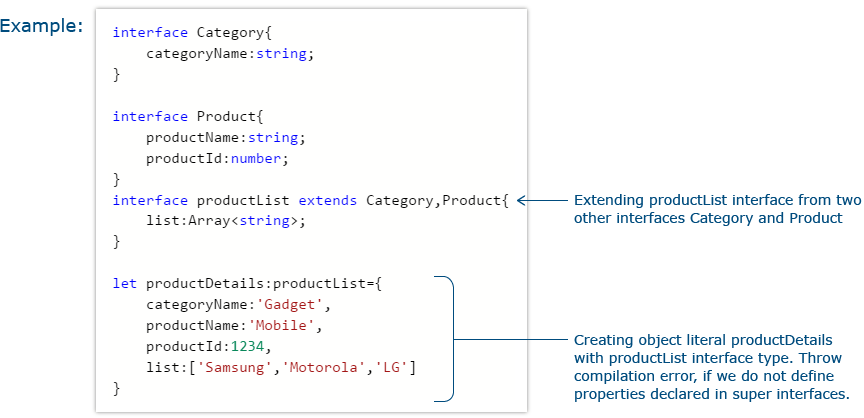
const prodObject = {productId: 1001, productName: 'Mobile', productCategory: 'Gadget'};

const productDetails: string = getProductDetails(prodObject); //this code works as typescript uses duck typing to match interfaces as properties exits it will not throw error but if we access productCategory inside function like we accessed productName then we will get compiler error that the productObj:Product doesn’t have that property.

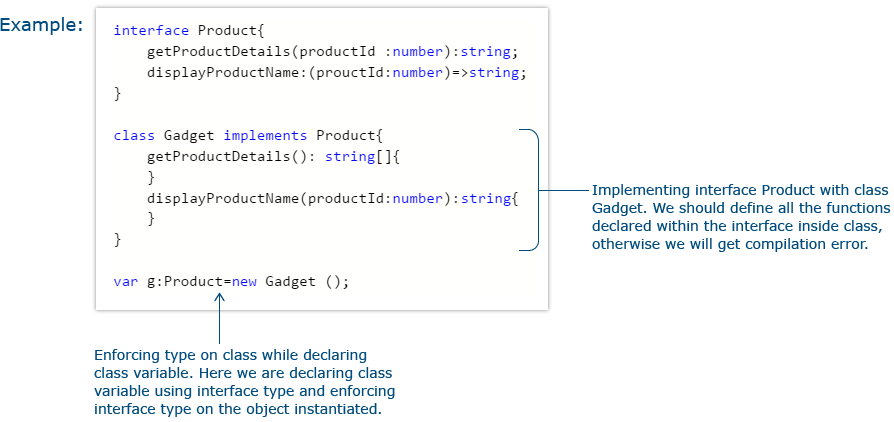
We can make function structure to be fixed and implemented by interface. We can fix the parameters it takes and type of return. This is how core functions are declared.



Multiple interfaces can be combined using extends keyword

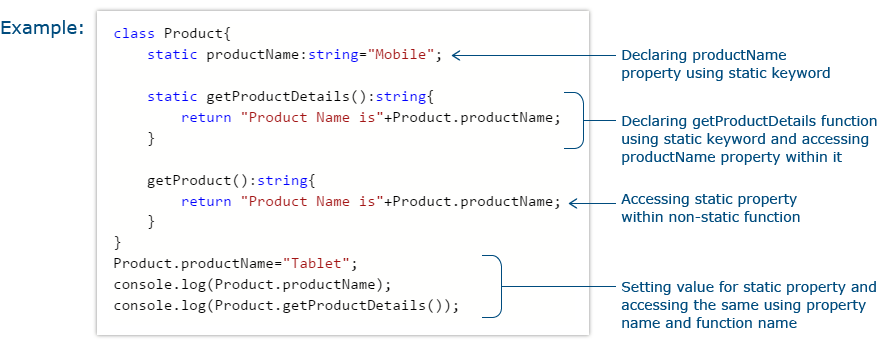


We can also enforce class to have some fixed properties by using interface and implements keyword.



Class is a template from which objects can be created, it provides behaviour and state storage.

Everytime we create new class the constructor gets invoked assigning the values, the parameters can be optional inside constructor, it had to be only one per class. All properties are bonded to object or instance of class created, Static keyword is used to declare class variable or method



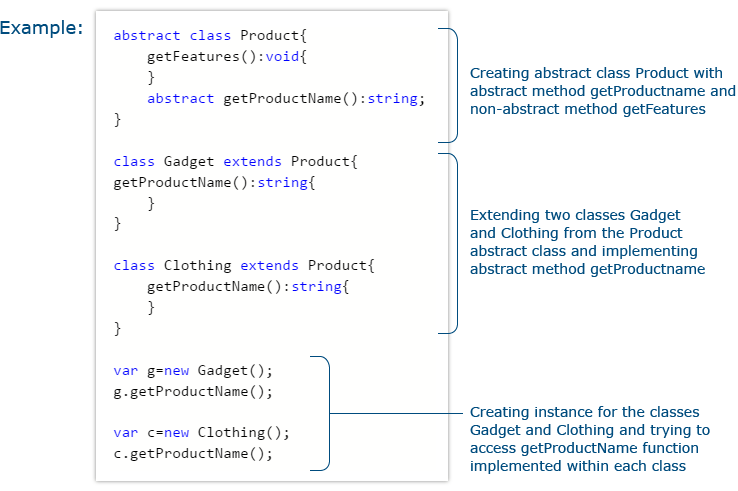
Instead of declaring access modifier and then assigning value inside constructor directly do it both in constructor like eg: class Product{ constructor(private prodId:number) }//this.prodId gets parameter

Inheritance is applied using extends keyword to make child class extend parent class, super() is used inside child constructor to invoke parent constructor. We use super.property or () to access parent properties. If no constructor is given in child class then it will directly refer parent constructor.

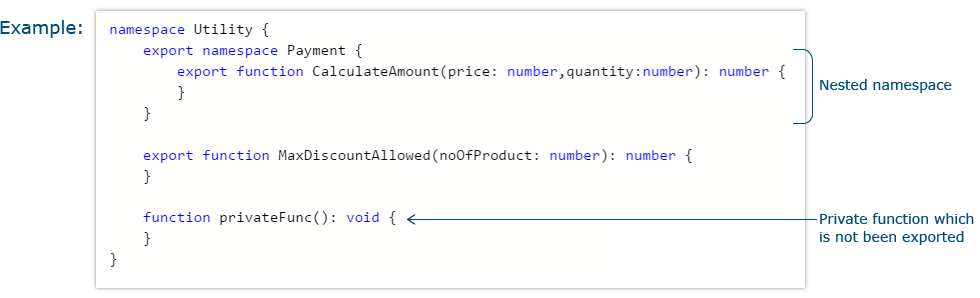
To access the private properties outside, typescript provides get and set keywords which will allow us to create getter and setter functions like below. Once created we can directly access them like how we access other properties. It will call automatically set if value is given or assigned to that property and invokes get if property is called.

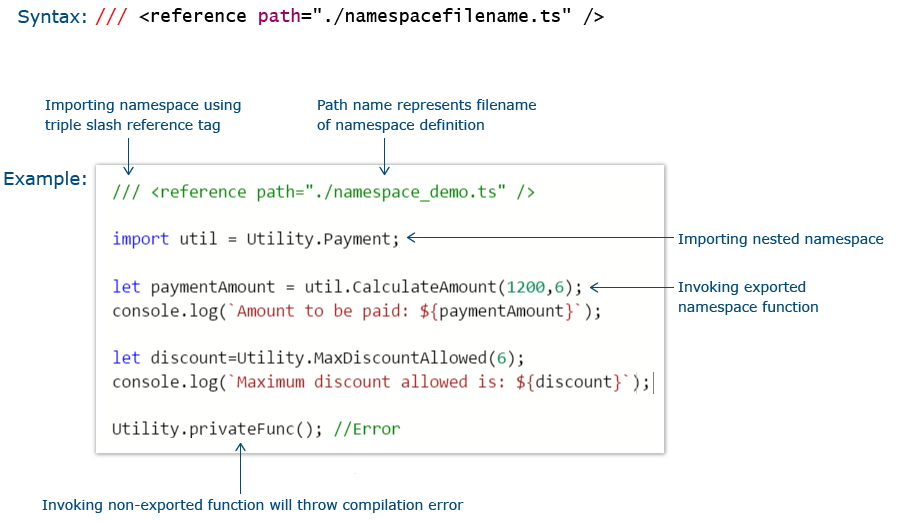


Abstract classes are base classes which may not be instantiated but makes child class to implement just like interface. Has to mention abstract keyword before class and also before properties which we want to implement abstract. Abstract properties cannot be private

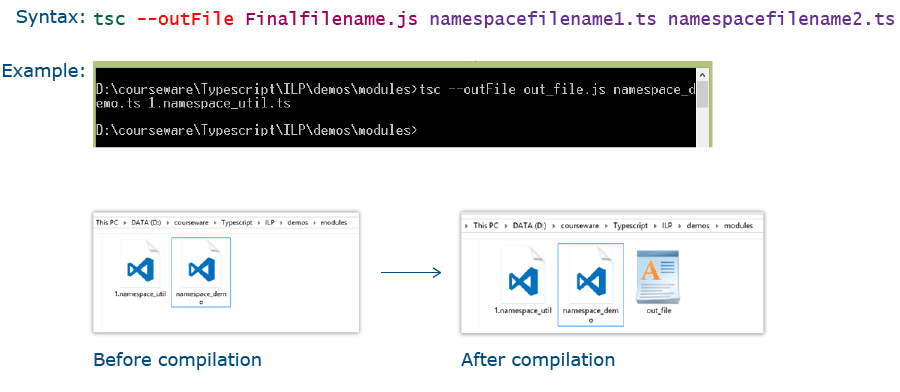


To export properties, we declare them all inside **namespace**. Here we created namespace Utility and it also had sub namespace Payment (export). We can import them in another file and get access to exported properties here. If it is not exported like privateFunc() then we can’t import in another file

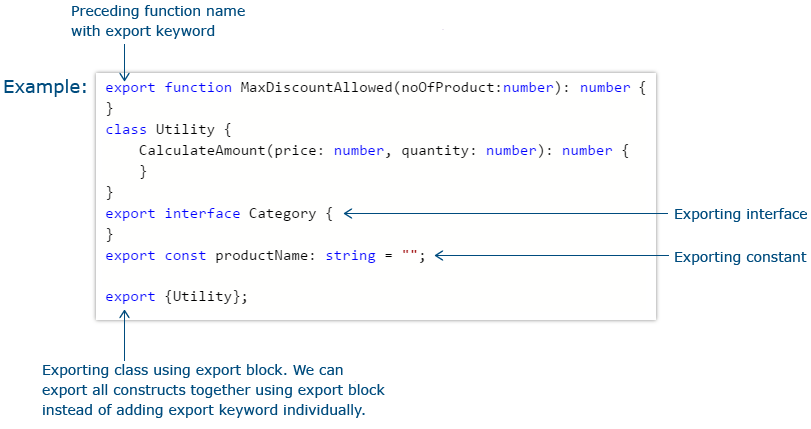


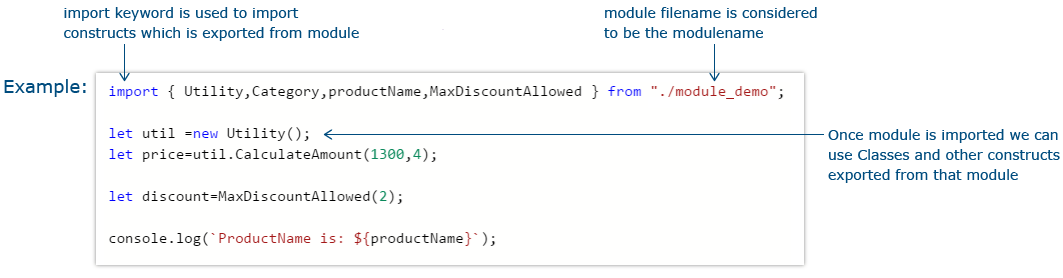


Since one file is dependent on another file importing it, have to transpile together.



Another way of exporting is Modules

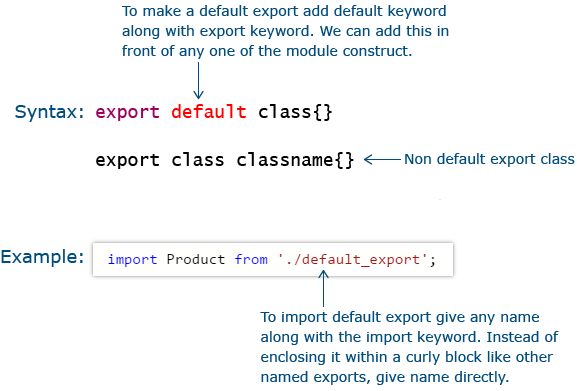




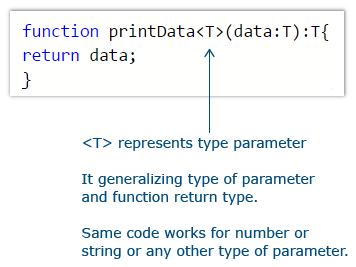
We can also use alias in both export { Utility as U} or import {U as myName}, import \* as myName ….

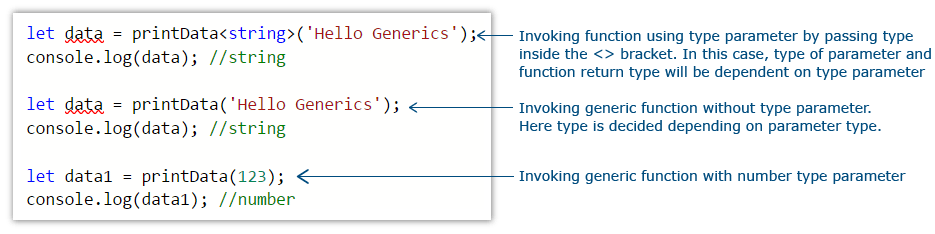
Namespace is for small scale applications where as Modules is for large scale apps, Modules need module loader which loads all the dependency modules into this environment.

On demand functionalities can be loaded by Modules, which is known as lazy loading. By using this feature while executing our application all the declared modules are not need to load at that moment, it only loads needed modules which are needed by the user to render initial look of the application on first load. Due to this concept performance of the application can be enhanced as the initial startup time of the application automatically decreases.



To make same code work for multiple data types we use Generics. Instead T we can give any char

 T is type parameter, always listed inside < >. T or any char



function orderDetails<P>(arg: Array<P>): Array<P> { // But T is used coding readability

console.log(arg.length); //this fn takes Type parameter and array of that type parameter

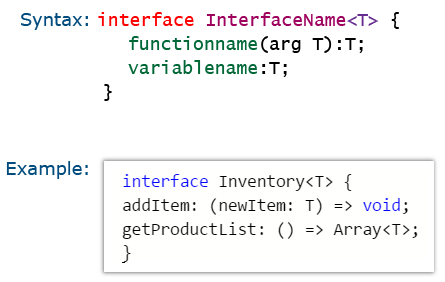
return arg;

}

const idList = orderDetails([101, 102, 103, 104]);

const nameList = orderDetails(['footwear', 'dress', 'cds', 'toys']); //both works

Generic Interface works with multiple data types





Generic classes

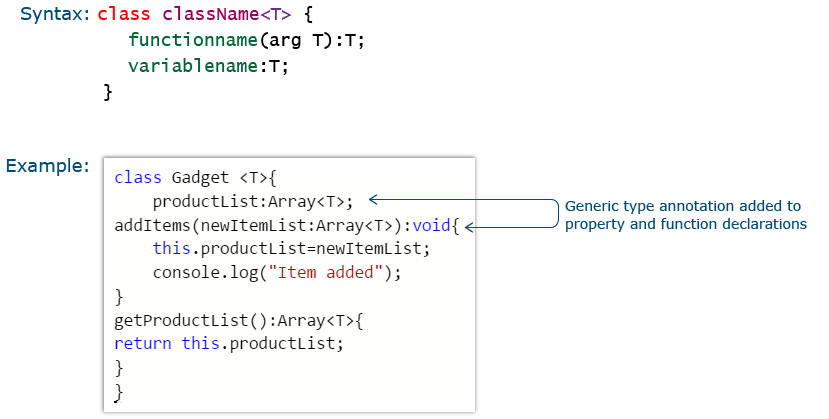
// Creation of reference variable of Gadget class with String array parameter

const product = new Gadget<string> ();

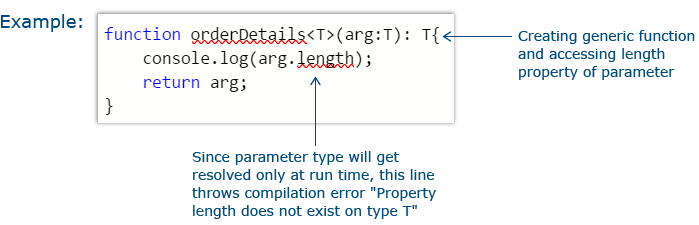
const productList: Array<string> = ['Mobile', 'Tablet', 'Ipod'];

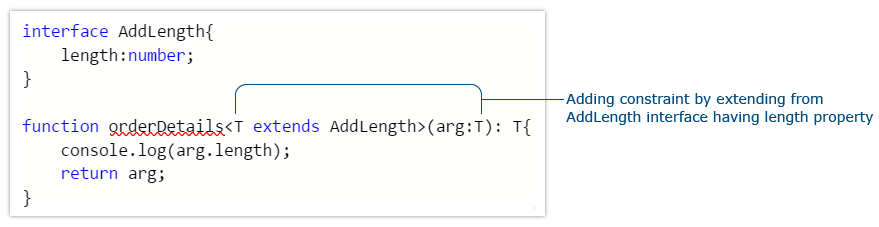
// logic to add product list value to product object

product.addItems(productList);

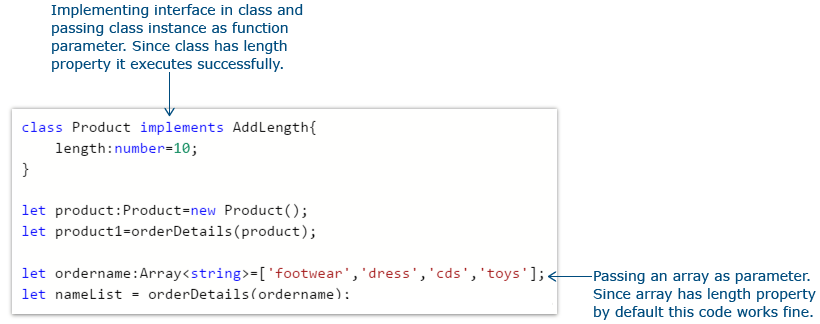


Generic constraints, to solve below problem constraints (extends that property) are used.





The objects passed into this generic type should have extended property like below.

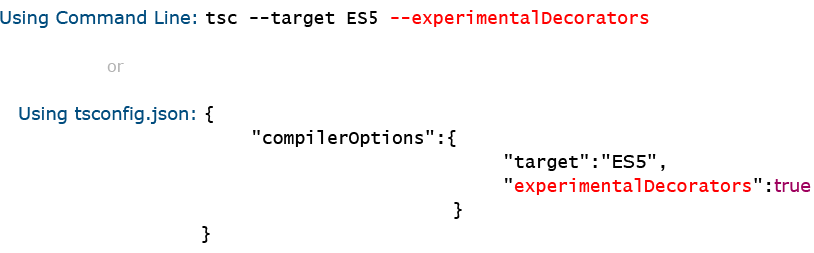


# Decorators

* Decorator is used to provide metadata and they are used to specify extra behavior of a class, method or property of a class.
* Decorators are used for declarative programming.
* Decorators are used to implement cross-cutting concerns.

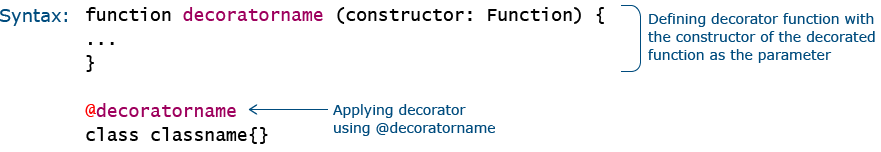
Decorators uses @expression form, where expression can represents a business logic which would be called at runtime with needed information respectively.

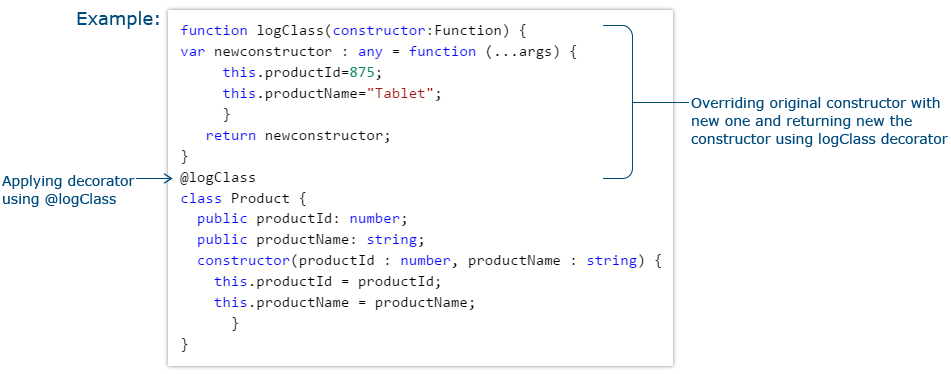
Developers need to enable experimentalDecorators true either in CLI or tsconfig.json



A Class Decorator are used just before a class declaration.

* It can be used to modify, observe or replace any class definition.
* The class decorator are applied to constructor of user defined class.
* The class decorator at runtime overrides original constructor logic with new one, returns only argument.
* We can log, modify or replace the original constructor within the class decorator function.





Here the newConstructor will override the constructor inside Product class eg: var p = new Product(999,”Phone”); console.log(p.productId, p.productName); This will give 875, “Tablet” in output cause the 999,Phne values are override by the newConstructor values.

The method decorator takes 3 parameters (in class it takes constructor), target is {}, propertyKey means function name i.e calculateAmountPayable string format, descriptor is thefunction description, so descriptor.value gives the total function which is applied by passing parameters as args and printed result inside decorator here.

function logMethod() {

return (target, propertyKey: string, descriptor: PropertyDescriptor) => {

return {

value: ( ... args: any[]) => { //args[] has all the parameters

console.log('Arguments: ', args.join(', '));

const result = descriptor.value.apply(target, args);

console.log('Total Payable Amount is: ', result);

return result;

} };};}

class Product {

@logMethod()

calculateAmountPayable(price: number, quantity: number) {

return price \* quantity;

}}

const p: Product = new Product();

p.calculateAmountPayable(220, 3); In log: Arguments:  220, 3 ; Total Payable Amount is:  660

Note: method decorator is invoked like @logMethod() but class decorator @invoke without ()

If we have a tsconfig.json file like below and runs the command **tsc** then it will compile and generate

{

"compilerOptions": {

"target": "es5",

"outDir": "Outfile",

"sourceMap": true,

"watch":true

},

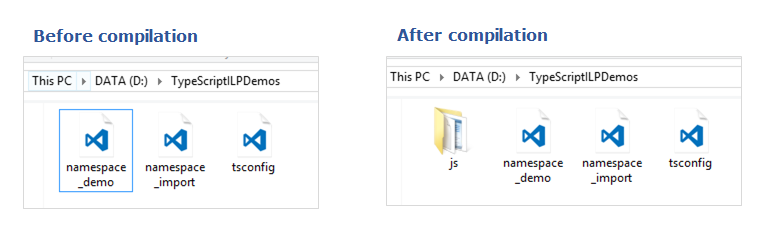
"files": [

"Namespace\_demo.ts",

"Namespace\_import.ts"

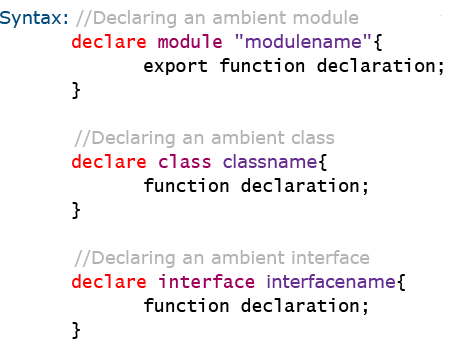
]

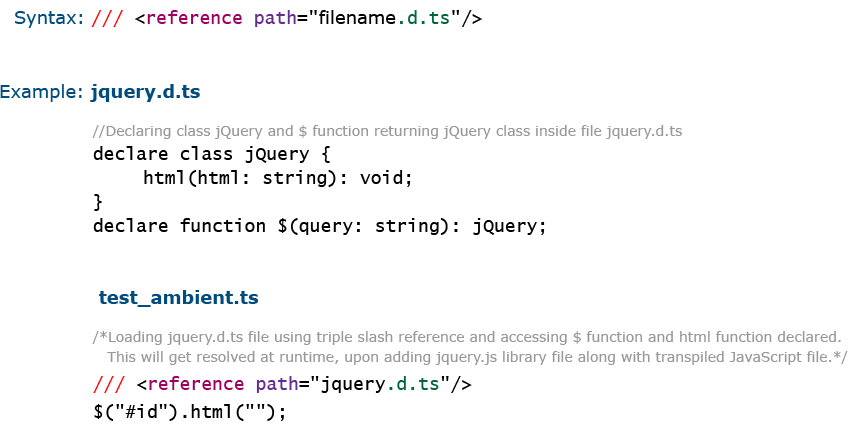
}



**Type Definitions**: Type definition files are the files that contain type information for a JavaScript file or JavaScript library to be used with in TypeScript code.

* Type definition files has the extension.d.ts
* They have declaration of functions, classes, modules, etc which is defined with in a JavaScript file be used with in TypeScript code.
* Ambient declarations are used to create type definition files for a specific JavaScript library or a JavaScript file
* Ambient declaration is basically used to tell TypeScript that we are trying to describe a code that exists elsewhere and get resolved at runtime.
* They are a kind of contract we make with the compiler. If these do not exist at runtime and we try to use them then the execution breaks down.





* Typings is used to find and download type definition files.
* We can get the type definitions from multiple sources like npm, definitelytyped, git , etc.

Npm I -g typings



**Union data types : mixed data types by | like below (ProdCountType is type alais)**

type ProdCountType = number | string | undefined; //we can define our custom types by **type**

function getProductDetail(pName: string, pCount: ProdCountType){

console.log(`Name: ${pName} ; Count: ${pCount}`)

}

getProductDetail("SamsungGalaxy10", 2);

getProductDetail("LenovoNote3", "No stock!");

getProductDetail("Redmi5", undefined);

**Function data types**

type PromoCodeGenerator = (pName: string, pId: number) => string;

let generator: PromoCodeGenerator = function (pName: string, pId: number): string {

return pName.substr(0, 4).toLocaleUpperCase() + pId;

}

let offer = generator("Lenovo 3", 1002);

console.log(`Please use the Promocode: ${offer}`)

**Array types**

type ProductStatusObject = { pId: string, availablity: boolean }

type AvailabiltyArray = [ ProductStatusObject ]

let product1: ProductStatusObject = { pId: "LENOV9912", availablity: true };

let product2: ProductStatusObject = { pId: "REDMI8878", availablity: false };

let product3: ProductStatusObject = { pId: "SAMSU5633", availablity: false }

let product4: ProductStatusObject = { pId: "OPPO1128",availablity: true}

let products: AvailabiltyArray = [product1];

products.push(product2,product3, product4);

let availableProducts = products.filter( (product) => product.availablity)

console.log(availableProducts)

**Type Literals (fixed static value)**

type ratings = "Good" | "Average" | "Excellent" | 0;

let customerFeedback1: ratings = "Good";

let customerFeedback2: ratings = 0;

let customerFeedback3: ratings = "Not bad"; // Error: Could not find literal value match.

Just like how we combine interfaces using extends keyword, we can combine type aliases using &

Ex: type Result = Score & Grade

Custom directive

import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';

@Directive({

selector: '[appRepeat]'

})

export class RepeatDirective { //viewContainer acc HTML container to add/remove elements

constructor(private \_templateRef: TemplateRef<any>, private \_viewContainer: ViewContainerRef) { } //templateRef gives <ng-template> content

@Input() set appRepeat(count: number) { //setter method assigns input to count value

for (var i = 0; i < count; i++) {

this.\_viewContainer.createEmbeddedView(this.\_templateRef);

}

}

}

<h3>Structural Directive</h3>

<p \*appRepeat="5">I am being repeated 5 times...</p>

We can use exportAs to access different properties of directives

import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';

@Directive({

selector: '[appRepeat]',

exportAs: 'repeat,changeText'

})

export class RepeatDirective {

constructor(private \_templateRef: TemplateRef<any>, private \_viewContainer: ViewContainerRef) { }

repeatElement(count: number) {

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

this.\_viewContainer.createEmbeddedView(this.\_templateRef);

}

}

changeElementText(count: number) {

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

document.getElementsByTagName('p').item(i).innerHTML = 'Text is changed...';

}

}

}

<h3>Structural Directive with exportAs property</h3>

<ng-template appRepeat #rd="repeat" #ct="changeText"> //we can use either rd or ct to

<p>I am being repeated...</p> access all properties

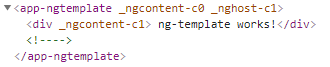
</ng-template>

<button (click)="rd.repeatElement(5)">Repeat Element</button>

<button (click)="ct.changeElementText(5)">Change Text</button>

<div> ng-template works!</div>

<ng-template>Inside ng-template tag</ng-template>



ng-template is a virtual element and its contents are displayed only when needed (based on conditions).

ng-template should be used along with structural directives like [ngIf],[ngFor],[NgSwitch] or custom structural directives.That is why in the above example the contents of ng-template are not displayed. So to display content inside ng-template there has to be a directive with asterisk (\*) like \*ngFor, \*ngIf or customer directive like above

Attribute directives

...

export class AppComponent {

colorName = 'red';

fontWeight = 'bold';

borderStyle = '1px solid black';

}

<p [ngStyle]="{

color:colorName,

'font-weight':fontWeight,

borderBottom: borderStyle

}">Demo for attribute directive ngStyle</p>

...

export class AppComponent {

isBordered = true;

isColor = true; }

<div [ngClass]="{bordered: isBordered, color: isColor}"> //classes defined in css

Border {{ isBordered ? "ON" : "OFF" }}

</div>

**Another example for custom directive:** This will change text and colour on click

<h3>Attribute Directive</h3>

<p [appMessage]="myMessage">Click Here</p>//myMessage variable in component.ts

import { Directive, ElementRef, Renderer2, HostListener, Input } from '@angular/core';

@Directive({

selector: '[appMessage]'

})

export class MessageDirective {

@Input('appMessage') message: string;

constructor(private el: ElementRef, private renderer: Renderer2) { //Renderer2 will set styles

renderer.setStyle(el.nativeElement, 'cursor', 'pointer'); //to make it clickable

} //this.el.nativeElement.focus() sets focus

@HostListener('click') onClick() {

this.el.nativeElement.innerHTML = this.message;

this.renderer.setStyle(this.el.nativeElement, 'color', 'red');

}

}

We can use exportAs property inside same directive, give exportAs:’changeMessage’ inside directive

<h3>Attribute Directive with exportAs property</h3>

<div [appMessage]="myMessage" #msg="changeMessage">

<p (click)="msg.onClick()">Click Here</p> //remove @HostListener, onClick invoked directly

</div>

For attribute binding <td [attr.colspan]="value"> First </td>

<button [style.font-size.px] = "isValid ? 3 : 6">Hello</button>

To work [(ngModel)] work import FormsModule

**Custom Pipes**

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

@Component({

selector: 'app-root',

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

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

})

export class AppComponent {

sortoption: string = "";

productsList: any = [

{ productName: 'Samsung J7', price: 18000 },

{ productName: 'Apple iPhone 6S', price: 60000 },

{ productName: 'Lenovo K5 Note', price: 10000 },

{ productName: 'Nokia 6', price: 15000 },

{ productName: 'Vivo V5 Plus', price: 26000 }

];

}

<h2> Sorting Products list using custom pipe </h2>

Sort the list based on

<select [(ngModel)]="sortoption">

<option value="prodName">Product Name</option>

<option value="price">Price</option>

</select><br/><br/>

<table border="1">

<thead>

<tr>

<th>Product Name</th>

<th>Price</th>

</tr>

</thead>

<tbody>

<tr \*ngFor="let products of productsList | sortPipe:sortoption">//cause in pipes we give extra parameters by : ,also value is taken before | directly

<td>{{products.productName}}</td>

<td>{{products.price}}</td>

</tr>

</tbody>

</table>

Custom pipe:

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({

name: 'sortPipe'

})

export class SortPipe implements PipeTransform {

transform(value: string[], args?: string): string[] { //custom pipe should implement transform()

if (args === 'prodName') {

return value.sort((a: any, b: any) => {

if (a.productName < b.productName) {

return -1;

} else if (a.productName > b.productName) {

return 1;

} else {

return 0;

}

});

} else if (args === 'price') {

return value.sort((a: any, b: any) => {

if (a.price < b.price) {

return -1;

} else if (a.price > b.price) {

return 1;

} else {

return 0;

}

});

}

return value;

}

}

**ExportAs in componenets**

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

@Component({

selector: 'app-courses-list',

templateUrl: './courses-list.component.html',

styleUrls: ['./courses-list.component.css'],

exportAs: 'courselist'

})

export class CoursesListComponent {

coursesList = [

{ courseId: 1, courseName: 'Node JS' },

{ courseId: 2, courseName: 'Typescript' },

{ courseId: 3, courseName: 'Angular' },

{ courseId: 4, courseName: 'React JS' }

];

course: any[];

changeCourse(name: string) {

this.course = [];

for (let i = 0; i < this.coursesList.length; i++) {

if (this.courses[i].courseName === name) {

this.course.push(this.courses[i]);

}

}

}

}

 <table border="1" \*ngIf="course">

<thead>

<tr>

<th>Course ID</th>

<th>Course Name</th>

</tr>

</thead>

<tbody>

<tr \*ngFor="let c of course">

<td>{{c.courseId}}</td>

<td>{{c.courseName}}</td>

</tr>

</tbody>

</table>

In app.component.html

<h2> Popular Courses </h2>

Select a course to view

<select #course (change)="cl.changeCourse(course.value)"> //invoking child comp function

<option value="Node JS">Node JS</option>

<option value="Typescript">Typescript</option>

<option value="Angular">Angular</option>

<option value="React JS">React JS</option>

</select>

<br/>

<br/>

<app-courses-list #cl="courselist"></app-courses-list> //#cl is reference to exportAs property

The above process has replace sending input (selected course value) to child compoenent.

**@Input**

...

export class CoursesListComponent {

...

course: any[];

@Input() set cName(name: string) { //setter method for cName

this.course = [];

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

if (this.courses[i].courseName === name) {

this.course.push(this.courses[i]);

}

}

}

}

...

Select a course to view <select #course (change)="name = course.value">// In parent component

<option value="Node JS">Node JS</option>

<option value="Typescript">Typescript</option>

<option value="Angular">Angular</option>

<option value="React JS">React JS</option>

</select><br/><br/>

<app-courses-list [cName]="name"></app-courses-list> //passing value to child variable cName

**@Output**

...

export class CoursesListComponent {

@Output() onRegister = new EventEmitter<string>(); //child comp

...

register(courseName: string) {

this.onRegister.emit(courseName);

}

}

<table border="1">

...

<tbody>

<tr \*ngFor="let course of courses">

<td>{{course.courseId}}</td>

<td>{{course.courseName}}</td>

<td><button (click)="register(course.courseName)">Register</button></td>

</tr>

</tbody>

</table>

<h2> Courses List </h2> //parent html

<app-courses-list (OnRegister)="courseReg($event)"></app-courses-list>

<br/><br/>

<div \*ngIf="message">{{message}}</div>

courseReg(courseName: string) {

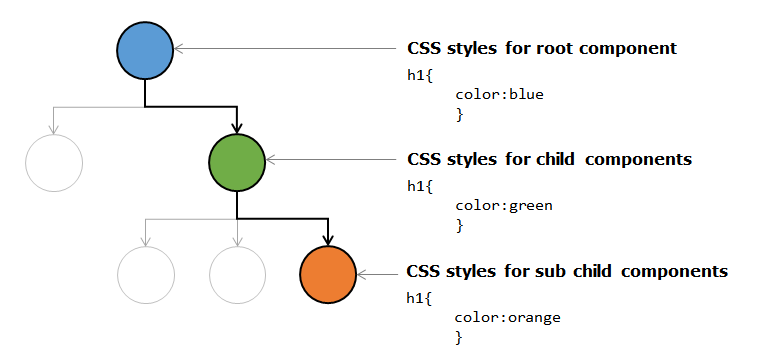
this.message = `Your registration for ${courseName} is successful`;

}

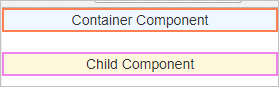
**Shadow DOM**

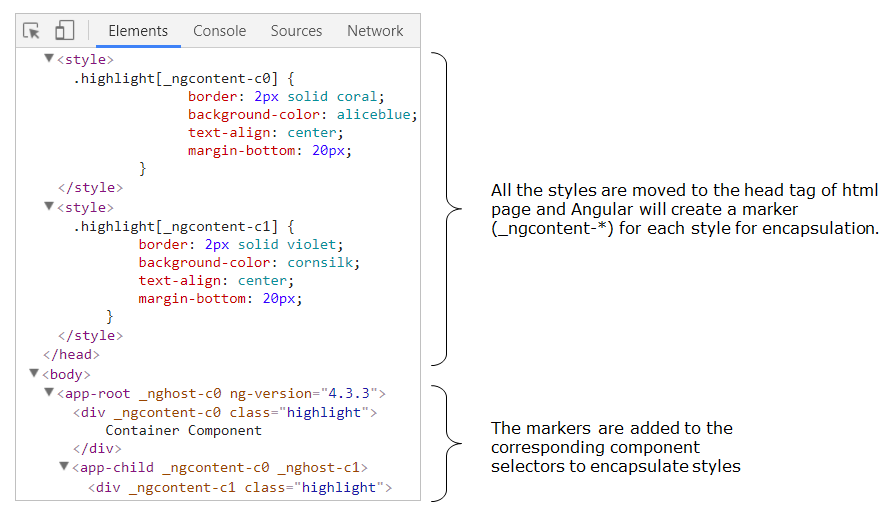
Shadow DOM is a web components standard by W3C. It enables encapsulation for DOM tree and styles. It means that shadow DOM hides DOM logic behind other elements and also confines styles only for that component.

Shadow DOM applies styles and encapsulates data for each comp, so that it doesn’t apply entire app



If we apply css style class with same name in parent and child components, angular shows difference





Markers are added to divs \_ngcontent-c0 which is also tagged to respective css class

Take an example of three components one inside another and all have same class, first 2 classes are marked and kept inside header style tag but the third one is kept by creating shadow root <app-second> and all styles are encapsulated inside that tag (look below image). This is called View encapsulation. To make thins like this we had to enable ViewEncapsulation

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

@Component({

selector: 'app-second',

templateUrl: './second.component.html',

styleUrls: ['./second.component.css'],

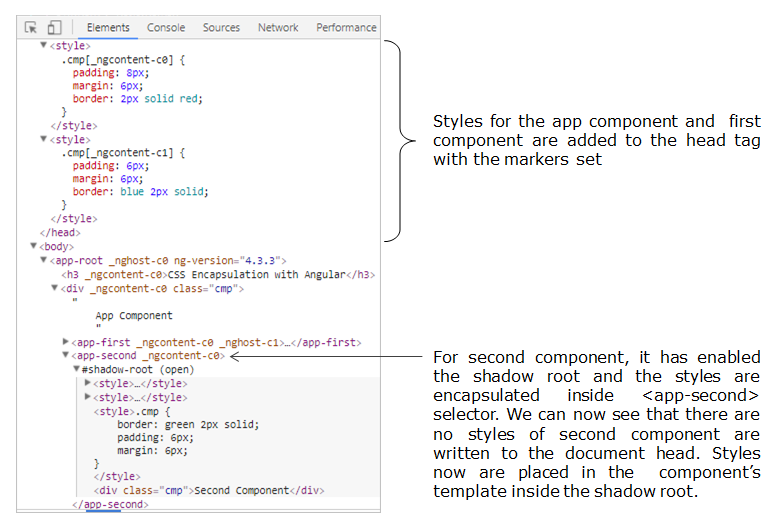
encapsulation: ViewEncapsulation.ShadowDOM

})

export class SecondComponent {

}

Parent and first components using default Shadow DOM, second comp using ViewEncapuslation.ShadowDOM



If we keep ViewEncapsulation.None then default shadow DOM also not applied and styles are used on entire application not just in that component, overriding each other. Not preffered.

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

@Component({

selector: 'app-root',

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

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

encapsulation: ViewEncapsulation.None

})

export class AppComponent {

}

**Life Cycles** (follows below order)

| **Interface** | **Hook** | **Support** |
| --- | --- | --- |
| OnChanges | ngOnChanges | Directive, Component |
| OnInit | ngOnInit | Directive, Component |
| DoCheck | ngDoCheck | Directive, Component |
| AfterContentInit | ngAfterContentInit | Component |
| AfterContentChecked | ngAfterContentChecked | Component |
| AfterViewInit | ngAfterViewInit | Component |
| AfterViewChecked | ngAfterViewChecked | Component |
| OnDestroy | ngOnDestroy | Directive, Component |

**Lifecycle Hooks**

* ngOnChanges – It will be invoked when Angular sets data-bound input property i.e., the property attached with @Input(). This will be invoked whenever input property changes its value
* ngOnInit – It will be invoked when Angular initializes the directive or component
* ngDoCheck -  It will be invoked for every change detection in the application
* ngAfterContentInit – It will be invoked after Angular projects content into its view
* ngAfterContentChecked – It will be invoked after Angular checks the bindings of the content it projected into its view
* ngAfterViewInit – It will be invoked after Angular creates component’s views
* ngAfterViewChecked – It will be invoked after Angular checks the bindings of the component’s views
* ngOnDestroy – It will be invoked before Angular destroys directive or component

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

@Component({

selector: 'app-child',

templateUrl: './child.component.html',

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

})

export class ChildComponent implements OnChanges {

@Input() title: string = 'I\'m a nested component';

ngOnChanges(changes) {

console.log('changes in child:' + JSON.stringify(changes));}}

in console- changes in child:{“title”:{“previousValue”:”old”,”currentValue”:”new”,firstChange:false}}

**Forms**

Import FormsModule to make template driven forms work

**Template driven forms**

<div class="container">

<div [hidden]="submitted">

<h1>Course Form</h1>

<form (ngSubmit)="onSubmit()" #courseForm="ngForm">

<div class="form-group">

<label for="id">Course Id</label>

<input type="text" class="form-control" required [(ngModel)]="course.courseId" name="id" #id="ngModel">

<div [hidden]="id.valid || id.pristine" class="alert alert-danger">

Course Id is required

</div>

</div>

...

**ngForm**

* ngForm is a built-in directive which will have an instance of each form element created in the application
* It has its own features which will get added to form element on the page
* name attribute is mandatory when [(ngModel)] is used on form elements
* Each input element will be an instance of FormControl class which will get registered with the name attribute value
* ngModel is used to track the state and validity of an element using the following keywords



/\* Selects any element that is NOT a paragraph \*/

:not(p) {

color: blue;

}

updateOn can be set like this to prevent validation on every keystroke

<form (ngSubmit)="onSubmit()" #courseForm="ngForm" [ngFormOptions]="{ updateOn: 'submit' }">

<div class="form-group">

<label for="id">Course Id</label>

<input type="text" class="form-control" required [(ngModel)]="course.courseId"

[ngModelOptions]="{ updateOn: 'blur' }" name="id" #id="ngModel">

<div [hidden]="id.valid || id.pristine" class="alert alert-danger">

Course Id is required

</div>

</div>

<div class="form-group">

<label for="name">Course Name</label>

<input type="text" class="form-control" required [(ngModel)]="course.courseName"

[ngModelOptions]="{ updateOn: 'submit' }" name="name" #name="ngModel">

<div [hidden]="name.valid || name.pristine" class="alert alert-danger">

Course Name is required

</div>

</div>

……

</form>

**Model Driven or Reactive Forms**

Import ReactiveFormsModule for model driven or reactive forms, creates form-controls inside .ts or model and reacts to it.

<form [formGroup]="registerForm">

<div class="form-group">

<label>First Name</label>

<input type="text" class="form-control" formControlName="firstName">

<p \*ngIf="registerForm.controls.firstName.errors" class="alert alert-danger">This field is required!</p>

</div>

…

<div [hidden]="!submitted">

<h3> Employee Details </h3>

<p>First Name: {{ registerForm.get('firstName').value }} </p>

registerForm: FormGroup;

submitted:boolean;

constructor(private formBuilder: FormBuilder) { }

ngOnInit() {

this.registerForm = this.formBuilder.group({

firstName: ['', { updateOn: 'blur', validators: [Validators.required] }],…..

**Custom Validators**

For template driven forms:

ng g directive emailValidator

<div class="form-group">

<label for="email">Author Email</label>

<input type="text" class="form-control" required [(ngModel)]="course.email"

name="email" #email="ngModel" **validateEmail**>

<div \*ngIf="email.errors && (email.dirty || email.touched)" class="alert alert-danger">

<div [hidden]="!email.errors.required">Email is required</div>

<div [hidden]="!email.errors.emailInvalid">{{email.errors.emailInvalid}}</div>

</div>

</div>

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

import { NG\_VALIDATORS, FormControl, Validator } from '@angular/forms';

@Directive({

selector: '[validateEmail]',

providers: [

{ provide: NG\_VALIDATORS, useExisting: EmailValidator, multi: true }

]

})

export class EmailValidator implements Validator {

validate(control: FormControl): { [key: string]: any } {

const emailRegexp = /^([a-zA-Z0-9\_\-\.]+)@([a-zA-Z0-9\_\-\.]+)\.([a-zA-Z]{2,5})$/;

if (!emailRegexp.test(control.value)) {

return { "emailInvalid": "Email is invalid" };

}

return null;

}

}

 Line 2: Import NG\_VALIDATORS which is a provider with an extensive collection of validators

Line 7: Register EmailValidator directive with NG\_VALIDATORS so that Angular recognizes the role of the directive in the validation process. multi:true adds EmailValidator class to the existing list of validators available in NG\_VALIDATORS

For Model driven forms:

Inside formBuilder email:['',validateEmail]

function validateEmail(c: FormControl) {

let EMAIL\_REGEXP = /^([a-zA-Z0-9\_\-\.]+)@([a-zA-Z0-9\_\-\.]+)\.([a-zA-Z]{2,5})$/;

return EMAIL\_REGEXP.test(c.value) ? null : {

emailInvalid: {

message: "Invalid Format!"

}

};

}

<div class="form-group">

<label>Email</label>

<input type="text" class="form-control" formControlName="email">

<p \*ngIf="registerForm.controls.email.dirty && registerForm.controls.email.errors?.emailInvalid" class="alert alert-danger">{{registerForm.controls.email.errors?.emailInvalid.message}}</p>

</div>

Dependency Injection (DI) is a mechanism where the required resources will be injected into the code automatically.

Angular comes with an in-built dependency injection subsystem.

**Why Dependency Injection?**

* DI allows developers to reuse the code across application.
* DI makes the code loosely coupled.
* DI makes the application development and testing much easier.
* DI allows the developer to ask for the dependencies from Angular. There is no need for the developer to explicitly create/instantiate them.

@Injectable({

providedIn:'root' //providedIn property registers BookService at the root level  (app module).

}) //@Injectable() decorator makes the class injectable into application components.

export class BookService {}

We can also make service available to all by mentioning in declarations inside app.module.ts

If we want to limit service to only one or few components then mention inside that component

import { BookService } from './book/book.service'; //component.ts

@Component({

selector: 'app-root',

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

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

**providers: [BookService]**

})

**RxJS :** Reactive Extensions for JavaScript (RxJS) is a third-party library used by Angular team.

RxJS is a reactive streams library used to work with asynchronous streams of data.

Observables, in RxJS, are used to represent asynchronous streams of data. Observables are a **more advanced version of Promises** in JavaScript

getBooks(): Observable<Book[]> {

return this.http.get<Book[]>("http://localhost:3020/bookList").pipe(

tap(data => console.log('Data Fetched:' + JSON.stringify(data))),

catchError(this.handleError)); //pipe,tap,catchError imported from rxjs/operators

} //pipe lets us to define series of operators separated by comma, Tap operator (similar to do) is to execute some statements once a response is ready which is mostly used for debugging purposes and catchError operator is used to handle the errors.

export class BookService {

...

private handleError(err:HttpErrorResponse) {//HttpErrorRe..imported from @angulr/common/http

let errMsg:string='';

if (err.error instanceof Error) {

// A client-side or network error occurred. Handle it accordingly.

console.log('An error occurred:', err.error.message);

errMsg=err.error.message;}

else {

// The backend returned an unsuccessful response code.

// The response body may contain clues as to what went wrong,

console.log(`Backend returned code ${err.status}`);

errMsg=err.error.status;

}

return throwError(errMsg); //throwError imported from rxjs

}

}

updateBook(book: Book): Observable<any> { //put is for update request

const options = new HttpHeaders({ 'Content-Type': 'application/json' });

return this.http.put<any>('http://localhost:3020/update', book, { headers: options }).pipe(

tap(\_ => console.log(`updated hero id=${book.id}`)),

catchError(this.handleError)

);

}

updateBook(id, name) {

this.bookService.updateBook({'id': id, 'name': name})

.subscribe(hero => this.books = hero, //assigning updated value to books variable

error => this.errorMessage = <any>error));

}

**Retry:** Some DB requests or webservices give wrong data or errors due to network, availability issues or old cache etc. To solve this import retry from rxjs/operators and then pipe it specifying n times.

getBooks(): Observable<Book[]> {

return this.http.get<Book[]>('http://localhost:3020/bookLis').pipe(

retry(3),

tap(data => console.log('Data Fetched:' + JSON.stringify(data))),

catchError(this.handleError));

}

Till now we are getting only response body as data but sometimes crucial info lies in headers or other part of response which can be read by **observe**:’response’ inside http request.

getBooks(): Observable<HttpResponse<Book[]>> {

return this.http.get<Book[]>(this.booksUrl**,{observe:'response'}**).pipe(

tap(books => {console.log(JSON.stringify(books));console.log(observe)}),

catchError(this.handleError));

}



# Interceptors

As Http Requests are immutable, we need to clone them to modify and then return the modified response. This interceptor changes every occurrence of HTTP in the httpRequest to HTTPS and sends a header token along with the request.

...

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

...

import { Interceptor1 } from './book/book.interceptor';

@NgModule({

imports: [BrowserModule, HttpClientModule],

declarations: [AppComponent, BookComponent],

providers: [{

provide: HTTP\_INTERCEPTORS,

useClass: Interceptor1,

multi: true

}],

bootstrap: [AppComponent]

})

export class AppModule { }

// Wires-up Interceptor1 into the application by importing HTTP\_INTERCEPTORS token in the app module.

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

import { HttpEvent, HttpInterceptor, HttpHandler, HttpRequest } from '@angular/common/http';

import { Observable } from 'rxjs';

@Injectable()

export class Interceptor1 implements HttpInterceptor {

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

const authReq = req.clone({

headers: req.headers.set('Authorization', 'Password')

});

return next.handle(authReq);

}

}

The interceptor class should be injectable and define an intercept method to implement HttpInterceptor. The method takes two arguments, req is the request object and should be of type HttpRequest and next should be the handler that is of type HttpHandler, that uses the handle method to return HttpEvent observable type.

**Non-JSON Response**

Set responseType:’text’ as option to make it handle non JSON reponse cause by default its JSON

getBooks(): Observable<string> {

return this.http.get(this.txtUrl, { responseType: 'text' })

.pipe(tap(data => console.log(data.length)),

catchError(this.handleError));

}

Whatever the response related options we give inside http method after URL

**Routing**

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

import { RouterModule, Routes } from '@angular/router';

import { BookComponent } from './book/book.component';

import { DashboardComponent } from './dashboard/dashboard.component';

import { BookDetailComponent } from './book-detail/book-detail.component';

const appRoutes: Routes = [

{ path: 'dashboard', component: DashboardComponent },

{ path: '', redirectTo: '/dashboard', pathMatch: 'full' }, //pathMatch:’prefix’ will match path prefix

{ path: 'books', component: BookComponent }, pathMatch is mandatory for redirectTo

{ path: 'detail/:id', component: BookDetailComponent }

];

@NgModule({

imports: [

RouterModule.forRoot(appRoutes)

],

exports: [

RouterModule

]

})

export class AppRoutingModule { }

Angular component router belongs to @angular/router module. To make use of routing, we should import Routes, RouterModule classes

We need to configure the routes and the router will look for a corresponding route when a browser url is changed, so we use router wherever we want to navigate etc.

Routes is an array that contains all the route configurations. Then we need to pass this array to the RouterModule.forRoot() function in the application bootstrapping function

Pass the appRoutes array to forRoot method of RouterModule class to configure with the Router and add it to the imports property

<nav>

<a [routerLink]='["/dashboard"]' routerLinkActive="active">Dashboard</a>

<a [routerLink]='["/books"]' routerLinkActive="active">Books</a>

</nav>

<router-outlet></router-outlet>// routerLinkActive is to apply CSS class to make it look active

this.router.navigate([url, parameters]) 🡪 this.router.navigate(['/detail', book.id]);

ngOnInit() {

/\* const id = +this.route.snapshot.paramMap.get('id');

this.bookService.getBook(id)

.subscribe((book:Book) => this.book = book);\*/

this.sub = this.route.paramMap.pipe(switchMap((params: ParamMap) =>

this.bookService.getBook(+params.get('id')))).subscribe(book => this.book = book);

}

# Back

goBack() {

window.history.back();

}

ngOnDestroy() {

this.sub.unsubscribe(); //on destroy or close unsubscribe all the subscribes

}

# Lazy Loading

When an Angular application has a lot of components, it will increase the size of the application. In such a scenario, at some point, the application takes a lot of time to load.

To overcome this problem, we can go for asynchronous routing i.e, we need to load modules lazily only when they are required instead of loading them at the beginning of the execution

Lazy Loading has the following benefits:

1)Modules are loaded only when user requests for it

2)We can speed up load time for users who will be visiting only certain areas of the application

Consider we need to lazy load a bookComponent which is big in application & taking time to load, create a book-routing.ts module file in bookComponent folder and give empty path (relative path).

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

import { RouterModule, Routes } from '@angular/router';

import { BookComponent } from './book.component';

import { LoginGuardService } from '../login/login-guard.service';

const bookRoutes: Routes = [

{

path: '',

component: BookComponent,

canActivate: [ LoginGuardService ]

}

];

@NgModule({

imports: [ RouterModule.forChild(bookRoutes) ],

exports: [ RouterModule ]

})

export class BookRoutingModule{ }

Also create a booking.module.ts inside this folder and declare this book routing module inside this

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

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

import { BookComponent } from './book.component';

import { BookRoutingModule } from './book-routing.module';

@NgModule({

imports: [ CommonModule, BookRoutingModule ],

declarations: [ BookComponent ]

})

export class BookModule { }

Now in main routing module we will import that module upon path books making it load only until books path is invoked. Lazy loading and reconfig happens only once, when route requested first time

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

import { RouterModule, Routes } from '@angular/router';

import {BookComponent} from './book/book.component'

import { DashboardComponent } from './dashboard/dashboard.component';

import { BookDetailComponent } from './book-detail/book-detail.component';

import {LoginComponent } from './login/login.component';

const appRoutes: Routes = [

{ path: 'dashboard', component: DashboardComponent },

{ path: '', redirectTo: '/login', pathMatch: 'full' },

{ path: 'books', loadChildren: () => import('./book/book.module').then(m => m.BookModule)},

{ path: 'detail/:id', component: BookDetailComponent },

{path:'login', component: LoginComponent}

];

@NgModule({

imports: [

RouterModule.forRoot(appRoutes)

],

exports: [

RouterModule

]

})

export class AppRoutingModule { }

In tsconfig.ts mention "module": "esnext"

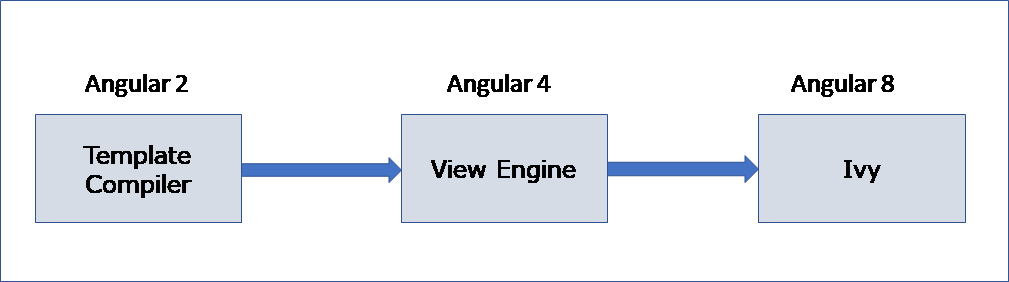
Once applied check on Network to see the time taken to load.

# Angular 9 and other Features

To migrate or upgrade from previous version of angular to latest

ng update @angular/cli @angular/core

Types of compilers in respective versions, In Angular 8 we need to mention explicitly and enable Ivy but from 9 it made default. Ivy is new 3rd gen compiler and renderer engine



Ivy removes unused code and makes bundle size less, Ivy compiles the components which is modified rather than entire app also faster build time

In tsconfig.app.json

"angularCompilerOptions": {

"enableIvy": false //disables Ivy

}

Angular CLI applications now compile in AOT mode by default, which helps to check template type-checking and fast rendering. AOT compiler converts HTML and TS into JS code at the build time itself, i.e., before the browser downloads and runs the code.

This provides faster rendering in the browser. Whereas JIT compiler compiles right when code is executed. In angular.json there is option “aot”:true, this is the reason we see errors on ng serve

tslib (npm i tslib) have some helpers like \_\_extends,\_\_rest etc. which will help in filling missing functionality when TS files are converted into JS

TestBed.configureTestingModule({

declarations: [ BookComponent],

providers: [ BookService]

});

let service = TestBed.get(BookService);

expect(service.getValue()).toBe('bookId:103');

From Angular 9, TestBed.get() is deprecated and it is replaced with TestBed.inject() which is type-safe.

The get method wasn't type-safe and it always used to return 'any' type and it is prone to errors.

TestBed.configureTestingModule({

declarations: [ BookComponent],

providers: [ BookService]

});

let service = TestBed.inject(BookService);

expect(service.getValue()).toBe('bookId:103');

Change in Angular Forms syntax:

In Angular 9, syntax of <ngForm> </ngForm>  is changed to <ng-form> </ng-form>

providedIn: Type<any> | 'root' | 'platform' | 'any' | null ; New scopes are added in Angular 9

To display dates, currencies etc. country specific Angular 9 improved internalization ng add @angular/localize

Angular 8 provides improved web worker bundling and one web worker is now included in every bundle built for production. As a result, it improves performance by providing parallelization.  ng generate webWorker <name>

Bazel support in Angular 8, Bazel is for betterment of build options,

Angular 7 asks CSS types, routing yes/no during project initialization

# Testing

To run the spec files, the command ng test will do the following:

1. It will refer the file – "angular.json" that redirects it to test.ts
2. test.ts specifies the file name pattern  to be targeted for testing - \*.spec.ts and executes the  karma command "karma start" for test spec execution
3. It now looks for karma.config which is present in same directory and executes all the specs.

Isolated unit tests are independent test like pipes etc. where no other modules etc. not required.

describe('WelcomeComponent', () => {

beforeEach(async(() => {

TestBed.configureTestingModule({

declarations: [WelcomeComponent],

}).compileComponents();

}));

it('should have title on the page', async(() => {

const fixture = TestBed.createComponent(WelcomeComponent);

const comp = fixture.debugElement.componentInstance;

expect(comp.pageTitle).toEqual('Welcome');

}));

});

TestBed creates an Angular testing module—an @NgModule class, It is the Angular testing utility that provides us the mock of the application’s root module: @NgModule

 createComponent is used to create an instance of the component-under-test.

TestBed: detectChanges

It is the Angular testing utility method that is responsible for detecting any angular change within a test. fixture.detectChanges()  helps test to convey to Angular when to perform the detection.

Mocked service and httpMock

import { async, TestBed, inject } from '@angular/core/testing';

import { HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';

import { Observable } from 'rxjs';

import { ProductService } from './product.service';

import { Product } from './product';

describe('ApiService', () => {

let httpMock: HttpTestingController;

let productService: ProductService;

beforeEach(async(() => {

TestBed.configureTestingModule({

// Providing HttpTesting module instead of original one

imports: [HttpClientTestingModule],

providers: [

ProductService

]

}).compileComponents();

// Getting the reference of all mocked services

httpMock = TestBed.inject(HttpTestingController);

productService = TestBed.inject(ProductService);

}));

// Checking service is injected successfully

it('should be created', inject([ProductService], (service: ProductService) => {

expect(service instanceof ProductService).toBe(true);

}));

// Checking for getProducts Method

describe('invoking getProducts method', () => {

// Creating variables to store results

let result;

let response: Product[];

let request;

beforeEach(() => {

// Assigning dummy value

productService.producttype = 'tablet';

// Creating mock response value

response = <Product[]>[];

// Invoking function

productService.getProducts()

.subscribe(data => {

result = data; // storing the mocked response under result

});

// Expecting request under particular url

request = httpMock.expectOne('assets/products/tablets.json');

// Flushing the mocked response

request.flush(response);

// Ensuring there is no standing calls

httpMock.verify();

});

// Checking the request is made under GET method

it('should make request under GET method', () => {

expect(request.request.method).toEqual('GET');

});

// Checking the result is populated with mocked value

it('should return value from backend', () => {

expect(result).toEqual(response);

});

});

});