CONTENTS

[JAVASCRIPT 7](#_Toc72004532)

[PRIMITIVE DATATYPES IN JAVASCRIPT 7](#_Toc72004533)

[DOM EVENTS IN JS 7](#_Toc72004534)

[INCLUDING SCRIPTS 9](#_Toc72004535)

[ARRAYS 10](#_Toc72004536)

[JAVASCRIPT EXECUTION 11](#_Toc72004537)

[OOPS IN JAVASCRIPT 14](#_Toc72004538)

[ENCAPSULATION 14](#_Toc72004539)

[ABSTRACTION 14](#_Toc72004540)

[INHERITANCE 15](#_Toc72004541)

[POLYMORPHISM 15](#_Toc72004542)

[CREATING JAVASCRIPT OBJECTS 15](#_Toc72004543)

[CONSTUCTOR PROPERTY OF AN OBJECT 16](#_Toc72004544)

[OBJECTS 17](#_Toc72004545)

[FUNCTIONS 19](#_Toc72004546)

[UNDERSTANDING THIS 24](#_Toc72004547)

[PROTOTYPE OBJECTS 24](#_Toc72004548)

[ES6 CONCEPTS 27](#_Toc72004549)

[VARIABLES 27](#_Toc72004550)

[STRING CONCATENATION – TEMPLATE LITERALS 28](#_Toc72004551)

[OBJECT LITERALS 28](#_Toc72004552)

[ARROW FUNCTIONS 28](#_Toc72004553)

[DESTRUCTURING 28](#_Toc72004554)

[DEFAULT PARAMETERS 29](#_Toc72004555)

[SPREAD OPERATOR 29](#_Toc72004556)

[REST PARAMETER 30](#_Toc72004557)

[FOR OF LOOP 31](#_Toc72004558)

[MAPS 31](#_Toc72004559)

[WEAKMAP 31](#_Toc72004560)

[SETS 31](#_Toc72004561)

[WEAKSET 32](#_Toc72004562)

[ASYNCHRONOUS JAVASCRIPT 32](#_Toc72004563)

[PROMISES 32](#_Toc72004564)

[PROMISES CHAINING 34](#_Toc72004565)

[ASYNC AND AWAIT 35](#_Toc72004566)

[CLASSES IN JAVASCRIPT 36](#_Toc72004567)

[SYNTAX 36](#_Toc72004568)

[INHERITANCE 37](#_Toc72004569)

[POLYMORPHISM 38](#_Toc72004570)

[HTML 5 WEB WORKER 39](#_Toc72004571)

[DEDICATED WEB WORKERS 39](#_Toc72004572)

[SHARED WEB WORKERS 41](#_Toc72004573)

[TYPESCRIPT 42](#_Toc72004574)

[TYPESCRIPT DEVELOPMENT ENVIRONMENT SET UP 42](#_Toc72004575)

[TYPESCRIPT TYPES 43](#_Toc72004576)

[TS FUNCTIONS 44](#_Toc72004577)

[IMPLICIT TYPING (TYPE INFERENCE) 44](#_Toc72004578)

[TYPESCRIPT CLASSES AND FUNCTION 44](#_Toc72004579)

[INHERITANCE 45](#_Toc72004580)

[GETTERS AND SETTERS IN CLASSES 46](#_Toc72004581)

[STATIC PROPERTIES AND METHOD 46](#_Toc72004582)

[ABSTRACT CLASSES 46](#_Toc72004583)

[SINGLETON AND PRIVATE CONSTRUCTORS 46](#_Toc72004584)

[INTERFACES 47](#_Toc72004585)

[MODULES IN TYPESCRIPT 47](#_Toc72004586)

[TYPESCRIPT COMPILER ARGUMENTS 48](#_Toc72004587)

[CREATING A TYPESCRIPT PROJECT 50](#_Toc72004588)

[CREATE A NODE PROJECT 50](#_Toc72004589)

[ANGULAR 10 52](#_Toc72004590)

[OVERVIEW 52](#_Toc72004591)

[SETTING UP THE DEVELOPMENT ENVIRONMENT 52](#_Toc72004592)

[CREATING ANGULAR 6 PROJECT 52](#_Toc72004593)

[FOLDER STRUCTURE 52](#_Toc72004594)

[ROOT COMPONENT IN DETAILS 53](#_Toc72004595)

[HOW ANGULAR APPLICATION IS BOOTSTRAPED 53](#_Toc72004596)

[CSS IN ANGULAR PROJECTS 54](#_Toc72004597)

[INSTALLING BOOTSTRAP CSS USING NPM 54](#_Toc72004598)

[ADDING IMAGE TO PROJECT 54](#_Toc72004599)

[CREATING THE FIRST CUSTOM COMPONENT 54](#_Toc72004600)

[COMPONENT SELECTORS 55](#_Toc72004601)

[DATA BINDING 56](#_Toc72004602)

[STRING INTERPOLATION - ONE WAY DATA BINDING 56](#_Toc72004603)

[PROPERTY BINDING - ONE WAY DATA BINDING 57](#_Toc72004604)

[EVENT HANDLING 57](#_Toc72004605)

[TWO WAY DATA BINDING [ngModel Directive] 58](#_Toc72004606)

[ANGULAR DIRECTIVES 58](#_Toc72004607)

[PRE-DEFINED DIRECTIVES 59](#_Toc72004608)

[ATTRIBUTE DIRECTIVES 60](#_Toc72004609)

[CUSTOM DIRECTIVES 60](#_Toc72004610)

[LOCAL REFEREENCE IN TEMPLATE 62](#_Toc72004611)

[COMMUNICATION BETWEEN COMPONENTS 63](#_Toc72004612)

[PARENT TO CHILD COMMUNICATION - PROPERTY BINDING AND @Input DECORATOR 63](#_Toc72004613)

[CHILD TO PARENT COMMUNICATION USING VIEWCHILD AND VIEWCHILDEN 66](#_Toc72004614)

[CHILD TO PARENT COMMUNICATION USING EVENTS - @Output 66](#_Toc72004615)

[@VIEWCHILD 67](#_Toc72004616)

[@VIEWCHILDREN 69](#_Toc72004617)

[VIEW ENCAPSULATION 71](#_Toc72004618)

[CONTENT PROJECTION 71](#_Toc72004619)

[CONTENT PROJECTION – ngContent 71](#_Toc72004620)

[MULTI SLOT CONTENT PROJECTION 72](#_Toc72004621)

[CONTENT PROJECTION - @ContentChild 73](#_Toc72004622)

[CONTENT PROJECTION - @ContentChildren 74](#_Toc72004623)

[ANGULAR LIFECYCLE HOOKS 74](#_Toc72004624)

[CREATING A SERVICE 75](#_Toc72004625)

[USING SERVICES 76](#_Toc72004626)

[HIERARCHICAL DEPENDENCY INJECTION 76](#_Toc72004627)

[INJECTING SERVICE INTO ANOTHER SERVICE 76](#_Toc72004628)

[CROSS COMPONENT COMMUNICATION USING SERVICES 77](#_Toc72004629)

[MAKING HTTP REQUEST 78](#_Toc72004630)

[BACKEND SET UP - SETTING UP FIREBASE 78](#_Toc72004631)

[POST REQUEST 79](#_Toc72004632)

[GET REQUEST 80](#_Toc72004633)

[FORMS 80](#_Toc72004634)

[TEMPLATE DRIVEN APPROACH 80](#_Toc72004635)

[ANGULAR MODULE 80](#_Toc72004636)

[ROUTING 81](#_Toc72004637)

[ROUTING TO MODULES 82](#_Toc72004638)

[ROUTING TO COMPONENTS 82](#_Toc72004639)

[SETTING UP ROUTE 82](#_Toc72004640)

[RELATIVE AND ABSOLUTE PATHS IN ROUTES 83](#_Toc72004641)

[NAVIGATING TO ROUTE PROGRAMATICALLY 83](#_Toc72004642)

[ROUTE PARAMETER PASSING IN ROUTE 83](#_Toc72004643)

[ROUTE OBSERVABLES 84](#_Toc72004644)

[QUERY STRING PARAMETERS AND FRAGMENTS - PASSING IN ROUTE 85](#_Toc72004645)

[CHILD ROUTES 85](#_Toc72004646)

[REDIRECTION AND WILD CHARACTERS IN ROUTE CONFIGURATION 87](#_Toc72004647)

[ROUTE GUARDS 87](#_Toc72004648)

[RXJS 88](#_Toc72004649)

[TERMINOLOGIES 88](#_Toc72004650)

[OBSERVABLES 88](#_Toc72004651)

[ANGULAR PIPES 91](#_Toc72004652)

[BUILT IN PIPES 92](#_Toc72004653)

[PARAMETERIZING AND CHAINING PIPES 92](#_Toc72004654)

[CREATING CUSTOM PIPES 93](#_Toc72004655)

[ANGULAR MODULES & OPTIMIZATION 94](#_Toc72004656)

[ANGULAR DEPLOYMENT 94](#_Toc72004657)

[ANGULAR UNIVERSAL 95](#_Toc72004658)

[WHAT IS ANGULAR UNIVERSAL? 95](#_Toc72004659)

[WHY ANGULAR UNIVERSAL? 95](#_Toc72004660)

[SETTING UP ANGULAR UNIVERSAL? 95](#_Toc72004661)

[SERVER AND CLIENT SIDE RENDERING AND RUNTIME INTERACTION 98](#_Toc72004662)

[ANGULAR UNIVERSAL – PRE- RENDERING 99](#_Toc72004663)

[APPLICATION SHELL 102](#_Toc72004664)

[STATE TRANSFER API 102](#_Toc72004665)

[REACT JS 103](#_Toc72004666)

[WHAT IS REACT? 103](#_Toc72004667)

[REACT CONCEPTS AND CREATING REACT COMPONENTS 103](#_Toc72004668)

[BASIC STEPS TO CREATE REACT COMPONENT 104](#_Toc72004669)

[OVERVIEW : CREATING REACT COMPONENT 104](#_Toc72004670)

[UNDERSTANDING JSX 104](#_Toc72004671)

[CREATING A DEMO REACT APP – BLOG POST APP 105](#_Toc72004672)

[COMMUNICATION BETWEEN COMPONENTS 106](#_Toc72004673)

[CODE IMPLEMENTATION 107](#_Toc72004674)

[CLASS BASED COMPONENTS 108](#_Toc72004675)

[BENEFITS OF CLASS BASED COMPONENT 108](#_Toc72004676)

[BUILD APP USING CLASS BASED COMPONENT 109](#_Toc72004677)

[CREATING CLASS BASED COMPONENTS 110](#_Toc72004678)

[LIFE CYCLE METHOD OF A CLASS BASED COMPONENT 111](#_Toc72004679)

[ADDING CSS TO COMPONENT 113](#_Toc72004680)

[OPERATORS 114](#_Toc72004681)

[FINAL APP DESIGN AND CODE 114](#_Toc72004682)

[HANDLING EVENTS AND USER INPUTS 115](#_Toc72004683)

[BINDING EVENTS 115](#_Toc72004684)

[HTTP REQUEST IN REACT 116](#_Toc72004685)

[NODE 117](#_Toc72004686)

[NODE ARCHITECTURE 117](#_Toc72004687)

[HOW NODE WORKS? 117](#_Toc72004688)

[SYNCHRONOUS AND ASYCHRONOUS NATURE OF NODE 118](#_Toc72004689)

[CREATING A NODE PROJECT 118](#_Toc72004690)

[NODE GLOBAL OBJECT 118](#_Toc72004691)

[NODE MODULES 118](#_Toc72004692)

[BUILD IN NODE MODULES 119](#_Toc72004693)

[EVENT MODULE 119](#_Toc72004694)

[HTTP MODULE 120](#_Toc72004695)

[FILE SYSTEM MODULE 121](#_Toc72004696)

[NODE PACKAGE MANAGER (NPM) 121](#_Toc72004697)

[PACKAGE.JSON 121](#_Toc72004698)

[INSTALLING A NODE PACKAGE 121](#_Toc72004699)

[USING THE NODE PACKAGE 122](#_Toc72004700)

[INSTALLING DEV DEPENDENCIES 122](#_Toc72004701)

[NPM PACKAGES AND SOURCE / VERSION CONTROL 122](#_Toc72004702)

[UN-INSTALLING PACKAGES 123](#_Toc72004703)

[PUBLISHING NPM PACKAGE TO NPM REPO 123](#_Toc72004704)

[ASYNC AND AWAIT IN NODE 123](#_Toc72004705)

[EXPRESS JS 124](#_Toc72004706)

[RESTFUL API USING EXPRESS JS 124](#_Toc72004707)

[NODEMON 124](#_Toc72004708)

[CONFIGURING LISTENING PORT 124](#_Toc72004709)

[ROUTING PARAMETER & HANDLING HTTP REQUESTS 124](#_Toc72004710)

[RESPONSE 126](#_Toc72004711)

[MIDDLEWARE / MIDDLEWARE FUNCTIONS 126](#_Toc72004712)

[SERVER-SIDE RENDERING USING TEMPLATING ENGINE 130](#_Toc72004713)

[EXPRESS GENERATOR 133](#_Toc72004714)

[DEBUG - EXPRESS 133](#_Toc72004715)

[REQUEST/RESPONSE OBJECT IN EXPRESS 134](#_Toc72004716)

[DYNAMIC ROUTE PARAMS 134](#_Toc72004717)

[WEB PACK 136](#_Toc72004718)

[MODULES IN JAVASCRIPT 136](#_Toc72004719)

[MODULE FORMATS/SYSTEM 137](#_Toc72004720)

[IMPLEMENTING COMMON JS MODULE FORMATS 137](#_Toc72004721)

[INSTALLING AND CONFIGURING WEBPACK 138](#_Toc72004722)

[INSTALLING AND CONFIGURING WEBPACK 138](#_Toc72004723)

[MODULE LOADERS IN WEBPACK 139](#_Toc72004724)

[ES6 MODULES SYSTEM 140](#_Toc72004725)

[CSS FILES IN WEBPACK 142](#_Toc72004726)

[CODE SPLITING USING WEBPACK 144](#_Toc72004727)

# JAVASCRIPT

### PRIMITIVE DATATYPES IN JAVASCRIPT

1. Number
2. String
3. Boolean
4. undefined
5. null

* JavaScript has dynamic typing- The datatypes are automatically assigned to the variable when the variable is assigned to it.
* The variable should not **start** with a number or special character except the ‘\_ ‘and ‘$’.
* The variable should not be reserved JS keyword.
* We can use a variable without explicitly declaring it.

|  |  |
| --- | --- |
| If the variable is used without any explicit declaration, they end up having global scope | If the variable is used with an explicit declaration, they end up having local scope |

#### STRICT MODE

* This prevents the use of undeclared variables.
* Restrict the use of reserved keywords as a variable name.
* Enabling strict mode will enable thowing of error rather than failing silently - when an exception occurs

#### TYPE COERSION

|  |  |
| --- | --- |
| var a=10, b=”10”;  a==b 🡪TRUE  a===b 🡪FALSE | “==” 🡪 in this comparison JavaScript do the automatic type conversion to compare the variables so it returns “TRUE”. This automatic type conversion is called type coersion  “===” 🡪 This compares the value and its type too. |
| **COMMON EXAMPLES OF TYPE COERSION (TYPE COERSION TO BOOLEAN)** | |
| For integers , any non-zero value is considered to be **TRUE**  var a=10; b =0;  var resultA = (a)? true:false;  var resultB = (b)? true:false;  console.log("a is "+resultA); 🡪 a is true  console.log("b is "+resultB); 🡪 b is false | For string , any non-zero length string is TRUE  var a='Hello'; b ='';  var resultA = (a)? true:false;  var resultB = (b)? true:false;  console.log("a is "+resultA); 🡪 a is true  console.log("b is "+resultB); ); 🡪 b is false |
| undefined && null  All “**undefined**” and “**null**” value is consider to be FALSE | var a; b =null;  var resultA = (a)? true:false;  var resultB = (b)? true:false;  console.log("a is "+resultA); 🡪 a is true  console.log("b is "+resultB); 🡪 b is false |

### DOM EVENTS IN JS

1. Event is some notable action for which script can respond.
2. An event handler is a code associated with a part of document and particular event

#### EVENT BINDING

|  |  |  |
| --- | --- | --- |
| **HTML** | **EVENT HANDLERS** | |
| <button id="clk">Click</button | function display(){ console.log("Display Me!"); } | function sayHello(){ console.log("Hello!");} |

##### EVENT BINDING USING ON

|  |  |
| --- | --- |
| var btnId = document.getElementById("clk"); | btnId onclick= display; |

##### EVENT BINDING USING ADD EVENT LISTENER

|  |  |
| --- | --- |
| **USING addEventListener():** Binds the handler with an event  var btnId = document.getElementById("clk");  btnId.**addEventListener**("click",display,false); | addEventListener() can bind multiple handlers with the same object .  btnId.addEventListener("click",display,false);  btnId.addEventListener("click",sayHello,false); |

##### EVENT UNBINDING USING ADD EVENT LISTENER

|  |  |
| --- | --- |
| **removeEventListener() : U**nbinds the event from the handler | btnId.**removeEventListener**("click",sayHello,false); |

##### EVENT PROPAGATION

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| The event propagation happens in 2 ways in the XHTML DOM   1. Event Capturing 🡪In capturing phase the event propagate happens from top to button in the DOM tree 2. Event Bubbling 🡪 In bubbling phase the event propagate happens from top to button in the DOM tree   **IMPLEMENTING EVENT CAPTURING AND BUBBLING**  We can implement event capturing and bubbling using  ***element*.addEventListener(*event, function, useCapture*);**   1. The first parameter is the type of the event 2. The second parameter is the function we want to call when the event occurs. 3. The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.    1. False 🡪 Event Bubbling 🡪 Bottom to Top    2. True 🡪 Event Capturing 🡪 Top to bottom | | | | |
| **HTML**  <div id="one"> 1  <div id="two"> 2  <div id="three"> 3  <div id="four">4  </div>  </div>  </div>  </div>   * The similar HTML DOM structure has been implemented as shown in the diagram * In the below code, we are binding the click event to all the divs | |  | | |
| **EVENT BUBBLING** | | | | |
| **JS**  document.getElementById("one").addEventListener(  "click", function(e) {  console.log("1");  }, false);  document.getElementById("two").addEventListener(  "click", function(e) {  console.log("2");  }, false);  document.getElementById("three").addEventListener(  "click", function(e) {  console.log("3");  }, false);  document.getElementById("four").addEventListener(  "click", function(e) {  console.log("4");  }, false); | **BOTTOM TO TOP** | | | |
| When we click ON 4 | |  | |
| When we click ON 3 | |  | |
| When we click ON 2 | |  | |
| **EVENT CAPTURING** | | | | |
| document.getElementById("one").addEventListener(  "click", function(e) {  console.log("1");  }, true);  document.getElementById("two").addEventListener(  "click", function(e) {  console.log("2");  }, true);  document.getElementById("three").addEventListener(  "click", function(e) {  console.log("3");  }, true);  document.getElementById("four").addEventListener(  "click", function(e) {  console.log("4");  }, true); | **TOP TO BOTTOM** | | | |
| When we click ON 4 | | |  |
| When we click ON 3 | | |  |
| When we click ON 2 | | |  |

#### EVENT OBJECT

|  |  |
| --- | --- |
| var btnId = document.getElementById("clk");  btnId.addEventListener("click",function(**evtObj**){  console.log(**evtObj**);  },false); | * The DOM events is always passed to the handler as an argument, which contains the details of the event. E.g target , currentTarget, type(eventType) etc.. |
| var btnId = document.getElementById("clk");  btnId.addEventListener("click",function(**evtObj**){  console.log(**evtObj.target**);#1  console.log(**evtObj.currenTarget**);#2  },false); | **#1 o/p** - <button id="clk">Click</button>  **#2 o/p** - <button id="clk">Click</button> |

#### EVENT METHODS

|  |  |  |
| --- | --- | --- |
| **stopPropagation() /**  **stopImmediatePropagation()**   * These method stops the event propagation further(bubbling/Capturing). Example here - is in case event capturing. * The event will not propagate further if it encounters either of the function * Both the function can be interchangeably used with one exception (explained below) | document.getElementById("one").addEventListener(  "click",  function(e) {  console.log("1");  }, true);  document.getElementById("two").addEventListener(  "click", function(e) {  console.log("2");  **e.stopPropagation();**  }, true);  document.getElementById("three").addEventListener(  "click", function(e) {  console.log("3");  }, true);  document.getElementById("four").addEventListener(  "click", function(e) {  console.log("4");  }, true); | o/P |
| **preventDefault()** 🡪The preventDefault() method cancels the event if it is cancelable, meaning that the default action that belongs to the event will not occur.  For example, this can be useful when:   1. Clicking on a "Submit" button, prevent it from submitting a form 2. Clicking on a link, prevent the link from following the URL   ***Note: The preventDefault() method does not prevent further propagation of an event through the DOM. Use the stopPropagation() method to handle this.*** | Example   * The default behavior of the hyperlink to capture the click event and open the hyperlink * This default behavior can stopped using the preventDefault()   **HTML**  <a id="wLink" href="https://www.w3schools.com/">w3schools</a>  **JS**  document.getElementById("wLink").addEventListener("click", function(e) {  e.preventDefault();  }); | |

### INCLUDING SCRIPTS

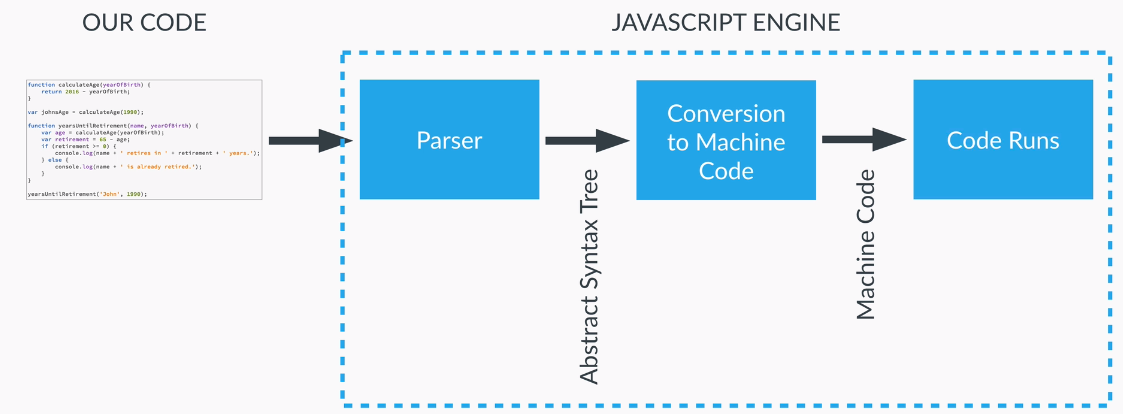
|  |
| --- |
| **NORMAL EXECUTION : <script src="script.js">**  Normal JavaScript Execution. HTML Parsing paused for script fetching and execution   1. When the HTML page loads , it starts the HTML parsing 2. If it encounters a script tag while parsing – the HTML parsing will pause 3. The script is fetched and followed by its execution, till that time HTML parsing will be blocked/paused 4. After the execution of script the HTML parsing resumes. |
| **USING ASYNC ATTRIBUTE : <script async src="script.js">**  Asynchronous JavaScript Execution. HTML parsing is paused only for the script execution   1. When the HTML page loads , it starts the HTML parsing 2. If it encounters a script tag while parsing – the script is fetched asynchronously along with HTML parsing. 3. Once fetched, the script starts executing. While the script is executing the HTML parsing will be blocked/paused 4. After the execution of script the HTML parsing resumes. |
| **USING DEFER ATTRIBUTE : <script defer src="script.js">**  Deferred JavaScript Execution. HTML parsing is never paused. Script execution happens after parsing is complete   1. When the HTML page loads, it starts the HTML parsing 2. If it encounters a script tag while parsing – the script is fetched asynchronously along with HTML parsing. 3. Once the script is fetched the HTML parsing resumes 4. The script will be executed only after HTML parsing is complete. |

### ARRAYS

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| * Stores the values of different data types unlike Java. * Arrays are JavaScript object   Example  var names=['John','Eric','Alex'];  console.log(names); |  | | | | | 1. Here the array’s index are property name and arrays element are the value to it 2. Similar to objects the arrays are access with property names e.g names["1"] 3. name[1] also works in arrays because of type coercion , which converts number into string | |
| **WAYS TO CREATE ARRAY** | | | | | | | | |
| var names =[“John”,”Mark”,”Emily”] | | | var years = new Array(2107,2018,2019); | | | | | |
| **ARRAY FUNCTION** | | | | | | | | |
| **INPUT** | | **FUNCTION** | | | | | | |
| var names =[“John”,”Mark”,”Emily”] | | Adding the last element  name[name.length]=”Joe” **OR** name.push(“Joe”); | | | | | | |
| **USING MAP**   1. The map() method iterates over all the elements and creates a new array from it , without actually modifying the actual arrays 2. If the associated call back function modifies the array element, the new array will store the modified value.   Note:   * map() does not execute the function for array elements without values. This method does not change the original array. | | | | | | | | |
| **EXAMPLE 1**  var arr = [10, 20, 30, 40];  var newArray = arr.map(function(element) {  element++;  console.log(element);  **return element;**  });  console.log(newArray);🡪 [11, 21, 31, 41];  **USING ARROW FUNCTION**  var newArray = arr.map(element => ++element); | | | | | **EXAMPLE 2**  var getAge = function(age) {  return 2020 - age;  };  var yearsOfBirth = [1982, 1992, 2004, 1957];  var listOfAdults = yearsOfBirth.map(function(element) {  return getAge (element);  });  console.log(listOfAdults); 🡪[**38,28,16,63]**  **USING ARROW FUNCTION**  var listOfAdults = yearsOfBirth.map(element => getAge(element)); | | | |
| names.pop() | | returns the last element and removing it from the array | | | | | | |
| names.shift() | | returns the first element and removing it from the array | | | | | | |
| names.push(element) | | Adds the element at the end of an array | | | | | | |
| names.unshift(element) | | Adds the element at the beginning of an array | | | | | | |
| const data = Array(5).**fill**(""); | | Filling arrays with empty string or some default value(passed in the fill method argument) | | | | | | |
| const data = ["1", "2", "3", "1", "4"];  console.log(Array.from(new Set(data))); | | Unique value from an array. | | | | | | |
| **LOOPING ARRAYS USING FOREACH**  var names=['John','Eric','Alex'];  names.forEach(function(value, index){  console.log("index="+index+" "+ "value="+value);  });  **USING ARROW FUNCTION**  var names = ['John', 'Eric', 'Alex'];  **names.forEach((value, index) => console.log(`index=${index} value=${value}`));** | | | | | | | **o/p**  index=0 value=John  index=1 value=Eric  index=2 value=Alex | |
| **ARRAYS TO OBJECT**  var user = ["Alex", "Doe", "XYZ Company", "Manager"];  var userObject = { ...user };  console.log(userObject); | | | | | | | **OBJECT TO ARRAYS**  var user = {  firstName: "Alex",  lastName: "Doe",  company: "XYZ Company",  designation: "Manager"  };  var userArray = Object.values(user);  console.log(userArray); | |
| **DEEP AND SHALLOW COPIES(ARRAYS)** | | | | | | | | |
| **DEEP COPIES** | | | | **SHALLOW COPIES** | | | | |
| Different variables point to different memory location. So changing will not change the other variables. | | | | Different variables point to same memory location. So changing on in-turn changes the other variables too. | | | | |

## JAVASCRIPT EXECUTION

**HOW JS CODE IS EXECUTED - OVERVIEW**



* Code executes in a Javascript engine of the browser
* The code is first parsed by a parser – if it has no syntax error it creates a data structure called Abstract Syntax Tree
* The DS then further translated into machine code which finally runs

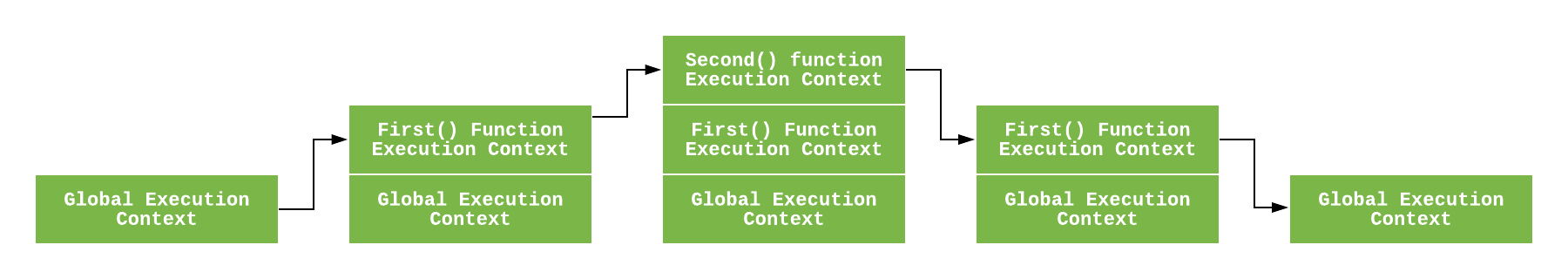
**EXECUTION CONTEXTS AND EXECUTION STACK**

|  |  |
| --- | --- |
|  | * All JS code runs in an environment – That environment is called Execution context. * The default execution context is **Global Execution Context.** * All the JS Code(variable/functions) which is not inside the function runs in **Global Execution Context** * The Global Execution Context code is associated with a global object called “window” object for the browser runtime environment.   var lastName="Hello";  **lastName === window.lastName** 🡨 true |

**EXECUTION STACK – WHEN AN EXECUTION CONTEXT IS CREATED?**

1. Execution stack, also known as “calling stack” in other programming languages, is a stack with a LIFO (Last in, First out) structure, which is used to store all the execution context created during the code execution.
2. When the JavaScript engine first encounters your script, it creates a global execution context and pushes it to the current execution stack. Whenever the engine finds a function invocation, it creates a new execution context for that function and pushes it to the top of the stack.The engine executes the function whose execution context is at the top of the stack. When this function completes, its execution stack is popped off from the stack, and the control reaches to the context below it in the current stack.

|  |  |
| --- | --- |
| let a = 'Hello World!';  function first() {  console.log('Inside first function');  second();  console.log('Again inside first function'); }  function second() {  console.log('Inside second function'); }  first(); console.log('Inside Global Execution Context'); | 1. When the code loads in the browser, the JavaScript engine creates a global execution context and pushes it to the current execution stack. When a call to first() is encountered, the JavaScript engines creates a new execution context for that function and pushes it to the top of the current execution stack. 2. When the second() function is called from within the first() function, the JavaScript engine creates a new execution context for that function and pushes it to the top of the current execution stack. 3. When the second() function finishes, its execution context is popped off from the current stack, and the control reaches to the execution context below it, that is the first() function execution context. 4. When the first() finishes, its execution stack is removed from the stack and control reaches to the global execution context. Once all the code is executed, the JavaScript engine removes the global execution context from the current stack. |



**HOW EXECUTION CONTEXT IS CREATED?**

|  |  |
| --- | --- |
| **OBJECT REPRESENTATION OF EXECUTION CONTEXT** | All execution context is always associated with an object like Global Execution context is associated with “window” object. The execution context object has 3 properties   1. **VARIABLE OBJECT(VO) :** This contains    1. Function arguments    2. Function Declaration    3. Inner variable declaration 2. **SCOPE CHAIN** 3. **“THIS” VARIABLE**   The execution context is created in 2 phases   1. **CREATION PHASE**    1. Creation of VO object    2. Creation of Scope Chain    3. Determine the value of “this” variable 2. **EXECUTION PHASE**    1. The code of the function that generated the current execution context is ran line by line |

**CREATION PHASE IN DETAILS**

Creation phase happens before the actual execution happens

**THE FIRST STEP: CREATION OF VO OBJECT**

* Code is scanned for **function declarations :** for each function a property is created in VO object , pointing to the function
* Code is scanned for **variable declarations :** for each variable a property is created in VO object , and set to undefined

**THIS IS CALLED HOISTING**

|  |  |
| --- | --- |
| **FUNCTION HOISTING**   * In case of function declaration – We can able to call the function before it is declared * This happens because the function declaration is hoisted during the Creation Phase itself. | calculateAge(1982);  function calculateAge(birthYear){  return 2020-birthYear;  } |
| **VARIABLE HOISTING**   * The variable can be accessed before declaring it as the variables are hoisted and set to undefined. | **EXAMPLE 1**  console.log(age);  var age = 23; |
| * The code outside the function and inside the function runs in different execution context. * In this code - The “age” variable is created– one in the Global Execution Context and another Function Execution context. * The code will access the respective variable depending upon the execution context they are executing. * #1 – Declares the age variable in Global execution context * #4 - Accessing the value from Global Execution Context * #2 – It is accessing the hoisted age variable which is in the execution context of the function its where it is undefined (not the Global Execution Context) * #3 – Accessing the age variable which is in the execution context of the function | **EXAMPLE 2**  var age= 30; **#1**  function foo(){  console.log("In foo before declaration="+age); **#2**  var age= 23;  console.log("In foo after declaration="+age); **#3**  }  foo();  console.log("Outside foo="+age); **#4** |

**THE SECOND STEP: SCOPING AND SCOPE CHAIN**

* The scoping answers a question – Where a variable can be accessed?
* Each function creates a scope. In JavaScript the only way to create new scope is to create a function.

**LEXICAL SCOPING –** JavaScript has lexical scoping, means a function inside another function can access to the scope of the outer function.

**SCOPE CHAIN**

|  |  |
| --- | --- |
|  | * The Scope chain propagate from bottom to top |

## OOPS IN JAVASCRIPT

The OOPS in any language has 4 pillars

1. **ENCAPSULATION**
2. **ABSTRACTION**
3. **INHERITANCE**
4. **POLYMORPHISM**

### ENCAPSULATION

|  |  |
| --- | --- |
| In encapsulation we group the related properties and methods that operate on those data together inside a wrapper called Object.  In procedural programming methods will have more number of parameters than object oriented. |  |
| **OBJECT ORIENTED WAY** | **PROCEDURAL WAY** |
| let employee = {  baseSalary : 3000,  overTime : 10,  rate :10,  getWage : function(){  return this.baseSalary+(this.overTime\*this.rate);  }  } | let baseSalary= 3000;  let overTime= 10;  let rate =10;  function getWage(baseSalary,overTime,rate){  return baseSalary+(overTime\*rate);  } |

### ABSTRACTION

Abstraction means hide the details and show/expose only the essentials. These exposed properties or methods can be accessed by client application or consumers.

**IMPLEMENTING ABSTRACTION: PRIVATE PROPERTIES AND METHODS**

|  |  |
| --- | --- |
| The problem with this implementation is that the property “radius” can be modified by client application, so the objective is to hide the “radius” property and expose only the draw() method to client application. This depicts **abstraction**.  There are two ways to achieve this   1. Using Local variables 2. Using setter & getters | function Circle(radius){  this.radius = radius;  this.draw = function(){  console.log("Draw Circle of Radius="+ this.radius);  }  }  let circle = new Circle(1);  circle.radius=2;  circle.draw(); |
| **USING LOCAL VARIABLES** | **SETTER AND GETTERS** |
| function Circle(rad){  let radius = rad;  this.getRadius = function(){  return radius;  }  }  let circle = new Circle(1);  console.log(circle.getRadius());  The local variable “radius” cannot be accessed/ modified directly by client application. It will be only be accessed with exposed method **getRadius()** | function Circle(){  let defaultRadius=0  Object.defineProperty(this,"**defaultRadius**",{  **get**:function(){  return defaultRadius;  },  **set**:function(value){  defaultRadius = value;  }  });  }  let circle = new Circle(1);  console.log(circle.defaultRadius);  circle.defaultRadius=10;  console.log(circle.defaultRadius); |
| Object.defineProperty(<object >,"**<*property\_to\_set*>**",{  **get**:function(){ // },  **set**:function(value){ // }  }); | 1. circle.defaultRadius 🡪The will call the “get” method of the **defaultRadius** property 2. circle.defaultRadius=10; 🡪 The will call the “set” method of the **defaultRadius** property and set its value to 10. |

### INHERITANCE

**INHERITANCE IS MADE POSSIBLE IN JAVASCRIPT USING PROTOTYPE OBJECT.** Refer Prototype Object section

### POLYMORPHISM

### CREATING JAVASCRIPT OBJECTS

|  |  |
| --- | --- |
| ***CREATING OBJECTS USING OBJECT LITERALS***   1. Objects literals in JavaScript are set of key value pairs. 2. The value can be another object or function   const circle = {  radius: 1, 🡨Property  location:{  x: 0,  y:0  },  draw:function(){ 🡨 Methods  console.log(“Draw Circle”);  }  };  **circle.draw(); //Calling function** | ***CREATING OBJECTS USING FACTORY FUNCTION***  function createCircle(radius){  **return** {  radius : radius,  draw: function(){  console.log(“Draw Circle of radius=”+ radius);  }  }  }  let circle1 = createCircle(1);  circle1.draw();  let circle2 = createCircle(2);  circle2.draw();   * **Here each object has own copy of properties and method** |

#### USING OBJECT.CREATE()

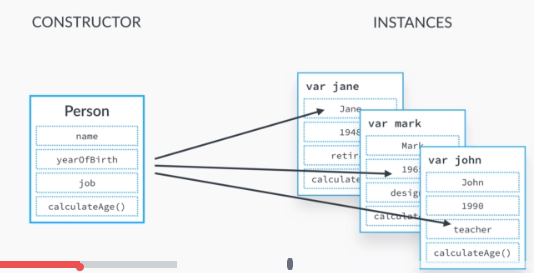
|  |  |
| --- | --- |
| const person = {  isHuman: false,  printIntroduction: function() {  console.log(`My name is ${this.name}. Am I human? ${this.isHuman}`);  }  };  **const me = Object.create(person);**  me.name = “Matthew”; // “name” is a property set on “me”, but not on “person”  me.isHuman = true; // inherited properties can be overwritten  me.printIntroduction();  // expected output: “My name is Matthew. Am I human? True” | **DETAILS ON OBJECT.CREATE()**   1. **const me = Object.create(person); 🡪 “me**” object will be created , where person will become the prototype object of **me** object. *Me.\_\_proto\_\_ ==person 🡺 true* 2. **me.name 🡪** This is a property of me object   **me.isHuman 🡪** Inherited property from person object |

#### CREATING OBJECTS USING CONSTRUCTOR FUNCTION

|  |  |
| --- | --- |
| function Circle(radius){  **console.log(this); 🡨 This points to empty object**  this.radius = radius;  this.draw = function(){  console.log(“Draw Circle of Radius=”+this.radius);  }  }  **let circle = new Circle(1);**  circle.draw(); | 1. By convention the first letter of constructor function should be capital. 2. We use “new” operator to create an object. **The new create an empty object and “this” points to the empty object** 3. In the constructor function returns the “this” object implicitly. |
| function Circle(radius) {  **console.log(this); 🡨 This points to global window object**  this.radius = radius;  this.draw = function() {  console.log(“Draw Circle of Radius=” + this.radius);  };  }  **let circle = Circle(1);** | When we a function Circle function constructor without new operator – this points to global window object |

##### CONSTRUCTOR FUNCTION IN DETAIL

1. In other programming languages like Java , we have a concept of classes , which act a template for objects created from that class/ template
2. The classes here are known as constructor which can be used to create multiple instances, as shown below.
3. The constructor act as a blueprint to create multiple instances. Every object created using the constructor function will have its own copy of properties and methods



1. The **function constructor** is the way by which we can create the blueprint

|  |  |
| --- | --- |
| **BLUEPRINT(FUNCTION CONTRUCTOR)** | **CREATING OBJECTS/INSTANCES** |
| var Person = function(firstName, lastName){  this.firstName = firstName;  this.lastName = lastName;  } | var john = **new** Person('john','Doe');  var mary = **new** Person('Mary','jain');  console.log(john);  console.log(mary); |
| When we use the “new” operator JS internally creates a “this” object and return the value at the end – as shown. | var Person = function(firstName, lastName){  **//var this ={} ;**  this.firstName = firstName;  this.lastName = lastName;  **//return this;**  } |

### CONSTUCTOR PROPERTY OF AN OBJECT

***EVERY OBJECT IN JS HAS A CONSTRUCTOR PROPERTY, WHICH REFERS TO THE FUNCTION WHICH IS USED TO CREATE THAT OBJECT.***

|  |  |
| --- | --- |
| **/\* Object Factory \*/**  function createCircle(radius){  return {  radius : radius,  draw: function(){  console.log(“Draw Circle of radius=”+ radius);  }  }  } | **/\* Object Factory \*/**    This Object uses an in-built **Object constructor function** |
| **let circle = createCircle(1);**  **circle.draw();**  **/\* Function Constructor \*/**  function Circle(radius){  this.radius = radius;  this.draw = function(){  console.log(“Draw Circle of Radius=”+this.radius);  }  }  **let anotherCircle = new Circle(1);**  **anotherCircle.draw();** | **/\* Function Constructor \*/**    **We can access the function itself using constructor function**  **anotherCircle.constructor(20);** |
| **/\* Object Literals \*/**  **let obj = {};** | **/\* Object Literals \*/** |

### OBJECTS

* Objects stores the different datatypes value as a key value pair

|  |  |  |
| --- | --- | --- |
| var person ={  firstName:’John’,  lastName: ‘Doe’,  birthYear:1982,  skills:[‘Java’,’C’]  }  **OR**  var person = new Object();  person.firstName=’John’; | **RETRIEVE THE VALUE FROM OBJECT**   * person.firstName 🡪 John * person[‘lastName’] 🡪Doe * var x =’birthYear’; person[x] 🡪1982 | **OBJECT EQUALITY**: Two objects considered to be equal when they pointing to same memory location.  Var myObj = {  prop: “property”  }  var myObJ2 = myObj;  console.log(myObJ2 === myObj); 🡪TRUE |
| **ADDING PROPERTY TO JS OBJECT**   * person.job =’designer’ OR * person[job]= ’designer’ |
| **OBJECT METHOD**  In JS object can have function called methods  var person ={  firstName:’John’,  lastName:’Doe’,  birthYear:1982,  calculateAge : function(){  return new Date().getFullYear() – this.birthYear;  }  }  console.log(person.calculateAge()); 🡨 Calling a function | | **SETTING AN OBJECT PROPERTY**  var person ={  firstName:’John’,  lastName:’Doe’,  birthYear:1982,  calculateAge:function(){  this.age = new Date().getFullYear() – this.birthYear;  }  }  person.calculateAge();  console.log(person); |
| **DELETING AN OBJECT PROPERTY**  var myObj = {  firstName: ‘John’,  middleName: ‘Eric’,  lastName:’Doe’  }  **delete myObj.middleName;**  **This deletes the middleName property from myObj object** | | **DYNAMIC OBJECT PROPERTY**  var company = “companyName”;  const user = {  name: “Alex”,  userid: “AL01”,  [company]: “XYZ Company”  }  console.log(user); |

#### RETRIEVING FROM OBJECTS

|  |  |  |
| --- | --- | --- |
| function Circle(radius){  this.radius = radius;  this.draw = function(){  console.log("Draw Circle of Radius="+this.radius);  }  }  let circle = new Circle(10);  We use dot or square bracket operator to retrieve object properties | | ***ADDING PROPERTY* : circle.location** ={x:10,y:10};  ***DELETING PROPERTY* : delete circle.location** ;  ***USING SQUARE BRACKET NOTATION***  The square bracket notation is used to retrieve the properties of an object   * When the property name is dynamic   let radiusProperty = 'radius';  console.log(circle[radiusProperty]);   * When property is not a valid JavaScript identifier   circle[‘**circle-radius’**]); 🡨 Special characters are not allowed in JS identifier |
| **ITERATING ALL OBJECT PROPERTIES** | for(let property **in** circle)  console.log(property); | |
| **FETCH ALL KEYS OF AN OBJECT** | Object.keys(circle); | |
| **CHECK A PROPERTY PRESENT?** | if('radius' **in** circle)  console.log("radius property exist"); | |
| **GET ALL KEYS AS ARRAY** | Object.keys(Circle) | |
| **GET VALUES AS ARRAYS** | Object.values(Circle) | |
| **GET OBJECT PROPERTIES AS ARRAY**  **(**each key -value pair in the object is an array**)** |  | |

**IMPORTANT POINTS**

* When we pass object in the function parameter – it will be always passed as reference type. So if the function makes any change in the passed object – it will change the original object. Unlike this, primitive types are passed as value type.

#### COPING EXISTING OBJECTS

|  |  |  |
| --- | --- | --- |
| **OBJECT ASSIGNMENT**  var Car= {  color:'Red',  brand:'BMW',  engine:{  make:2020,  power:'10 BP'  }  }  **let myCar = Car;**   * This creates a shallow copy of the existing object * The equality operator points to same memory location – So changing copied object will change the original object too. | **OBJECT. ASSIGN**  var Car= {  color:'Red',  brand:'BMW',  engine:{  make:2020,  power:'10 BP'  }  }  let carCopy = Object.assign({},Car);  console.log(carCopy);  carCopy.color = blue;  carCopy.engine.make ='Ferari';  console.log(Car);  **OUTPUT** | * Object.assign copies the source object to target object and finally return the target object.     **CONCLUSION**   * If we change the top-level property from copied object- it will not update that property in original object * Unlike that if we update the property of nested object property from copied object – it will update the original object also. |
| **SPREAD OPERATOR**  **let newCar = {...Car};**  This has same behavior as “object.assign()” | * DEEP COPY : In deep copy the objects are independent to each other and points to different memory location . * JSON.parse(JSON.stringify(Car)) 🡨 **CREATES THE DEEP COPY** | |
| **OBJECT.CREATE()** |  | |

### FUNCTIONS

* Function is an instance of object type that’s why function behaves like an object

|  |  |  |  |
| --- | --- | --- | --- |
| **WHY FUNCTIONS IN JAVASCRIPT ARE CALLED FIRST CLASS FUNCTION** | |  | |
| function Circle(radius) {  this.radius = radius;  this.draw = function() {  console.log(“Draw Circle of Radius=” + this.radius);  };} | | | * The Function in JS is an object in JavaScript.   **WHO CREATES FUNCTION OBJECT?**   * Circle is a function here and the constructor property return the function which has been used to create the object. * JS use the native Function constructor to create the Circle function as shown. |
| **PASSING FUNCTION AS PARAMETER**  var add = function(a,b){  console.log(a+b);  }  var subtract = function(a,b){  console.log(a-b);  }  var calculator = function (operand1 , operand2 ,operation){  operation(operand1,operand2);  }  **calculator(5,3,add);**  **calculator(5,3,subtract);** | **FUNCTION RETURNING A FUNCTION**  function calculatoController(operation) {  if (operation == “add”) {  return function add(op1, op2) {  return op1 + op2;  };  } else if (operation == “sub”) {  return function sub(op1, op2) {  return op1 – op2;  };  } else {  return function noOperation() {  console.log(“Operation No Supported”);  };  }  }  var addCalc = calculatoController(“add”);  console.log(addCalc(1, 2));  var subCalc = calculatoController(“sub”)(3, 1); 🡨 Can be called like this too  console.log(subCalc); | | |
| **IIFE**  **(**function(){  var a = 10;  var b= 20;  console.log(a+b);  }**)();**  console.log(a+b); 🡨 Here a ,b is not accessible | 1. IIFE stands for – Immediately invoke function expression. 2. IFFE creates only function scope variable (avoids creation of global variables) which get destroyed once the execution is over 3. It an anonymous function .We use if we want to execute logic right away on the page load without explicitly calling it. 4. This help in data privacy and avoid updating any variable in the global execution context. 5. This is called only once | | |
| **PASSING VALUE TO IIFE** | (function(op1, op2) {  console.log(op1 + op2);  })(1, 2); | | |

#### FUNCTION STATEMENTS & EXPRESSION

* Expression always returns some value. E.g [2+3 (o/p = 5) is an expression] so we expect a JS function to return something we should write an expression.

|  |  |  |
| --- | --- | --- |
| **FUNCTION EXPRESSION** | | **FUNCTION STATEMENTS** |
| FUNCTION EXPRESSION | ANONYMOUS FUNCTION EXPRESSION | function add(a,b){  return a+b;  }  console.log(add(1,2)); |
| var add = function **addFn**(a,b){  return a+b;  };  console.log(add(1,2)); | var add = function (a,b){  return a+b;  };  console.log(add(1,2)); |

#### FUNCTIONS IN OBJECT

|  |  |
| --- | --- |
| **LEGACY WAY** | **ES6 WAY** |
| const person = {  firstName: “John”,  lastName: “Doe”,  fullName: function() {  return this.firstName + “ “ + this.lastName;  }  };  console.log(person.fullName()); | const person = {  firstName: “John”,  lastName: “Doe”,  fullName() {  return this.firstName + “ “ + this.lastName;  }  };  console.log(person.fullName());  We can get rid of “**function**” keyword in ES6 |
| **GETTER & SETTERS – THIS HELPS IN IMPLEMENTING ABSTRACTION IN JS** | |
| const person = {  firstName: "John",  lastName: "Doe",  get fullName() {  return this.firstName + " " + this.lastName;  },  set fullName(name) {  let parts = name.split(" ");  this.firstName = parts[0];  this.lastName = parts[1];  }  };  console.log(person.fullName);  person.fullName = "Mike Smith"; 🡨 Call the setter  console.log(person.fullName); 🡨 Calls the getter | 1. **get && set** keywords are used to create getter and setters 2. When we set the value of a property it will call the setter of the property 3. When we access the property it calls the getter of the property 4. The **fullName** property can be accessed as a property not as a function |

#### EXCEPTION HANDLING IN JS

|  |  |
| --- | --- |
| function addNumber(op1 , op2){  if(isNaN(op1) || isNaN(op2))  **throw** "Paramter is not well defined!";  return op1 + op2;  }  window.addEventListener("load", function(){  try{  addNumber("**abc**",2);  }**catch**(exp){  console.log(exp);  }**finally**{  console.log("Method Executed");  }  }); | Exception scenario |

#### CALLING FUNCTION (UNDERSTANDING THIS)

1. **REGULAR FUNCTION CALL**
2. **FUNCTION CONSTRUCTOR.**
3. **CALLING FUNCTION IN CONTEXT TO AN OBJECT**
4. **USING “CALL”, “APPLY” AND “BIND“FUNCTION.**

|  |  |  |
| --- | --- | --- |
| **REGULAR FUNCTION CALL –** When the function called in a regular way. It gets called on the global object i.e. “window” object.  Example:  var print = function(){  console.log("Print Function Called");  console.log(this);  }  print(); | |  |
| **CALLING FUNCTION IN CONTEXT TO AN OBJECT**  Here “this” refers to the object on which function is called.  var myObj ={  prop :'propertyValue',  printProp: function(){  console.log(this);  }  }  myObj.printProp(); | |  |
| **FUNCTION CONSTRUCTOR**.  var foo =function(){  console.log(this);  }  **new foo();** |  | Whenever a function is called using **new** keyword, JS adds the highlighted lines internally, which is empty, so as the output.  var foo =function(){  //var this ={};  console.log(this);  //return this;  } |
| **USING CALL FUNCTION**   1. It can be used to invoke (call) a method with an owner object as an argument (parameter). 2. With call(), an object can use a method belonging to another object. 3. In this example we are assigning the **inflateTire** property (which happens to be a function) to mike object . 4. When **mike** object calls the **inflateTire** function as [mike.inflateTire();] , which hasbelow statement   **this.tirePressure = this.tirePressure + 3;**   1. But mike object has no **tirePressure** property so the value comes out to be **this.tirePressure + 3 = NaN**      1. To call the **inflateTire** we should have bicycle instance . To achieve this we have to call the method using “call()” function 2. **Call** method the object in the parameter on which we call the function   **mike.inflateTire.call(bicycle);** | | This example calls the fullName method of person, using it on person1:  var person = {  fullName: function() {  return this.firstName + " " + this.lastName;  }  }  var person1 = {  firstName:"John",  lastName: "Doe"  }  var person2 = {  firstName:"Mary",  lastName: "Doe"  }  **person.fullName.call(person1);** |
| **Example 2**  function Bicycle(name, speed, tirePressure) {  this.name = name;  this.speed = speed;  this.tirePressure = tirePressure;  this.inflateTire = function () {  this.tirePressure = this.tirePressure + 3;  }  }  function Mechanic(name) {  this.name = name;  }  var bicycle = new Bicycle("Hero", 10, 10);  var mike = new Mechanic("Mike");  mike.inflateTire = bicycle.inflateTire;  //mike.inflateTire();  **mike.inflateTire.call(bicycle);**  console.log(bicycle); |

#### CALL, APPLY AND BIND METHODS

***These methods help us to set the “this” variable manually***

|  |  |
| --- | --- |
| **CALL FUNCTION**   1. The call method helps in function borrowing i.e. An object borrows a function from another object. 2. Using the “call” function calls a function we can invoke the function on the object which we send in the call method parameter. 3. The “eimly” object borrows the “fullName” function from “john” object 4. Using call function we can invoke the fullName() on Emily object. | var john = {  firstName: "John",  lastName: "Doe",  fullName() {  return this.firstName + " " + this.lastName;  }  };  var eimly = {  firstName: "Eimly",  lastName: "Clark"  };  console.log(john.fullName());  **// function borrowing**  console.log(**john.fullName.call(eimly)**); |
| **SHARING FUNCTION AMONG OBJECTS USING CALL/ CALL METHOD WITH ARGUMENT**   1. In general scenario the functions are shared among the objects, but not the part of an object. 2. Using the **call** function we can call the “fullName()” function on the object which we pass in its parameter. We can pass 3. We can pass arguments also using call function. | var john = {  firstName: "John",  lastName: "Doe"  };  var eimly = {  firstName: "Eimly",  lastName: "Clark"  };  **function fullName(location) {**  **return this.firstName + " " + this.lastName + " " + location;**  **}** |
| console.log(**fullName.call(john, "England")**);  console.log(**fullName.call(eimly, "India")**); |
| **APPLY FUNCTION**   1. When we have more number of arguments in call function , so rather than sending them individually we can send it as an array 2. To send parameters as an array we need to use “**apply**” function. | var john = {  firstName: "John",  lastName: "Doe"  };  function fullName(location, state) {  return this.firstName + " " + this.lastName + " " + location + " " + state;  }  var eimly = {  firstName: "Eimly",  lastName: "Clark"  }; |
| **CALL USING APPLY**  console.log(fullName.**apply**(john, ["Laos", "US"]));  console.log(fullName.**apply**(eimly, ["Kolkata", "WB"])); |
| **BIND FUNCTION**   1. Bind function binds a function with an object 2. The bind function actually return the function with which it is bind to and then invoked later (unlike apply and call which are invoked immediately) | var john = {  firstName: "John",  lastName: "Doe"  };  function fullName(location, state) {  return this.firstName + " " + this.lastName + " " + location + " " + state;  }  var eimly = {  firstName: "Eimly",  lastName: "Clark"  };  let johnProfile = fullName.**bind**(john);  let eimlyProfile = fullName.**bind**(eimly);  console.log(johnProfile("Laos", "US"));  console.log(eimlyProfile("Kolkata", "WB")); |
| The **johnProfile** stores the function |
| **FUNCTION CURRYING**  **WHAT IS CURRYING?**  Currying is a process in functional programming in which we can transform a function with multiple arguments into a sequence of nesting functions. It returns a new function that expects the next argument inline.  **We can preset some of the argument of the function** | function checkShape(length) {  return function(breadth) {  return length == breadth ? "Square" : "Rectangle";  };  }  let lengthOfShape = checkShape(10); 🡨 Preseting the value (length of the shape)  console.log(lengthOfShape(5));  console.log(lengthOfShape(10)); |
| **FUNCTION CURRYING USING BIND**   1. We can also achieve function curing using bind method. 2. Here we are preseting the length value 3. Note - “this” is a window object(global Object) | function checkShape(length, breadth) {  return length == breadth ? "Square" : "Rectangle";  }  **let lengthOfShape = checkShape.bind(this, 10);**  console.log(lengthOfShape(5));  console.log(lengthOfShape(10)); |
| **FUNCTION CURRYING USING BIND ON CUSTOM OBJECT**  var john = {  name: "John",  speciesType(speciesType, gender) {  console.log(`${this.name} is ${speciesType} and ${gender}`);  }  };  var mary = {  name: "Mary"  }; | **CALLING**  let johnDetails = john.speciesType.bind(john, "Human");  let maryDetails = john.speciesType.bind(mary, "Human");  johnDetails("Male");  maryDetails("Female");   1. Here we preset the **speciesType** argument and passing the “gender” argument in subsiquent calls. |

## UNDERSTANDING THIS

## PROTOTYPE OBJECTS

**INHERITANCE CAN BE IMPLEMENTED IN JS USING THE PROTOTYPE OBJECT**

|  |  |  |
| --- | --- | --- |
|  | The prototype object can be considered as parent object.    **Here both object inherits the same object base i.e. they have same parent object i.e. prototype object** |  |
|  | * `The x and y object call all the methods it inherits from the parent prototype object. * Even in case of Arrays it has a prototype object – **That’s why we can call different functions like “indexOf” etc., which is actually defined in its prototype object.** |
| function Circle(radius) {  this.radius = radius;  this.draw = function() {  console.log("Draw Circle of Radius=" + this.radius);  };  }  let circle = new Circle(1);  let circle1 = new Circle(1);  console.log(**Object.getPrototypeOf(circle) === Object.getPrototypeOf(circle1)**);   * Here also all the circle object created (using new operator) from same constructor function will have same prototype object. * Any property / method defined in the prototype object is never becomes the part of circle object – They are accessed by circle object via inheritance. | |

|  |  |  |
| --- | --- | --- |
| function User(name) {  this.name = name;  this.greetUser = function () {  return "Hello ! " + this.name;  }  }  var mike = new User("Mike");  var alex = new User("Alex");  console.log(mike);  console.log(alex); |  |  |
| 1. In a typical object oriented programming language , the objects encapsulate the properties and shares the behavior(methods). The “methods” are never becomes the part of objects. 2. In JS when we create an objects using function constructor the functions and properties both becomes part of object (**as shown above**).Each object has their own copy of properties .To solve this issue we create function using **prototype** object. | | |

**ITERATING OBJECT PROPERTY**

|  |  |
| --- | --- |
| function Circle(radius) {  this.radius = radius;  this.draw = function() {  console.log("Draw Circle of Radius=" + this.radius);  };  }  let circle = new Circle(1); |  |
| **QUESTION: WHY WE CANNOT ABLE TO ITERATE PROPERTY DEFINED IN OBJECT’S PROTOTYPE?** | |
|  | |

#### PROTOTYPE OBJECTS

|  |  |  |  |
| --- | --- | --- | --- |
| * Prototype objects are used to create kind of **blueprint** from which we can create objects. * Function in JavaScript is an object. When a JS engine processes a function, it creates 2 objects   1. First - with the name as the function. That’s why functions can be accesses using its name.   2. Prototype object – These prototype object can be accessed using “prototype” property of the function object. | | |  |
|  | Function Object can be accessed using function name | **Function Prototype Object : foo.prototype** | |

#### PROTOTYPE OBJECT IN DETAILS

|  |  |
| --- | --- |
| function Foo(){ }  **Foo.prototype.prop="Hello World!”;**  var object1 = new Foo();  var object2 = new Foo();  console.log(object1.\_\_proto\_\_.prop); 🡪 “Hello World!”  console.log(object2.\_\_proto\_\_.prop); 🡪“Hello World!”  object1.\_\_proto\_\_ == object2.\_\_proto\_\_ 🡪 true  object1.prototype == object2.prototype 🡪true   * When we call a function using “**new**”, JS engine creates a property “\_\_proto\_\_”. This property is called **dunder proto** * We can access the function’s prototype object using dunder proto (\_\_proto\_\_) as show in diagram. |  |

#### SHARING VIA PROTOTYPE OBJECT

|  |  |  |
| --- | --- | --- |
| **PROPERTY SHARING** | | |
| 1. **The User function acts as a template for all the objects created from it.** 2. **User.prototype.greetUser =** This creates a function in the prototype object 3. All the objects created from User function constructor points to the same prototype object (highlighted) 4. If the property and functions are created in prototype object – A single copy of property / function will be shared by all the objects. 5. ***When we call a function on an object [mike.greetUser()] It will be looked up in the current object and if not found it is further looked up in prototype object. This is how the prototype properties are shared between the objects***. | | function User(name) {  this.name = name;  User.prototype.greetUser = function () {  return "Hello ! " + this.name;  }  }  var mike = new User("Mike");  var alex = new User("Alex");  mike.prototype == alex.prototype; 🡨 **TRUE** |
| When we add a property to a prototype object, it will be shared by all objects created using new operator from the function constructor.  function Foo(){ }  Foo.prototype.message ="Hello World!";  var Object1 = new Foo();  var Object2 = new Foo();  console.log(Object1.\_\_proto\_\_.message);  console.log(Object2.\_\_proto\_\_.message)  **OR**  console.log(Object1..message);  console.log(Object2.message);; |  | |
| **FUNCTION SHARING**  function FooBar(){ }  FooBar.prototype.printName =function(name){  return “Hello ”+ name;  };  var Obj1 = new FooBar();  var Obj2 = new FooBar(); | **CALING THE SHARED FUNCTION**  console.log(Obj1.printName("Alex"));  console.log(Obj2.printName("John"));  **OR**  console.log(Obj1.\_\_proto\_\_.printName("Alex"));  console.log(Obj2. .\_\_proto\_\_.printName("John")); | |

#### PROTOTYPE OBJECT - PROPERTY LOOK UP

|  |  |
| --- | --- |
| 1. When we access a property from an object. It do a look up in the object itself 2. If the object don’t have that property – it will do a look up in prototype object |  |
|  |

#### CONSTRUCTOR PROPERTY OF PROTOTYPE OBJECT

|  |  |
| --- | --- |
| The prototype object has a property “constructor” – from which we get hold of the function itself via the dunder proto object.  Step 1: Create a new Object  **var person = new Human();**  Step 2: Get the prototype object  **var proto = person.\_\_proto\_\_;**  Step 3: Access the function using constructor property on Prototype object  **var proto.constructor();** 🡨 This executes the function itself |  |

#### WHO CREATES THE PROTOTYPE OBJECT?

**OBJECT FUNCTION**

**Object function is a global function which gets called whenever an object is created without any specific constructor function**. For example –**var simple ={};**

|  |  |
| --- | --- |
|  | Accesing the Object Function |
|  | Calling the Object Function. Calling the Object function creates an empty object |
| **var objectVar = new Object();** | Creating an object on the Object Function |

|  |  |
| --- | --- |
| The dunder proto of the Object function’s prototype object is **NULL**. This terminates prototype chaining . |  |

|  |  |
| --- | --- |
| **PROTOTYPE CHAINING** | 1. When an object is created using new operator – The JS engine creates a prototype object of the constructor function 2. The Prototype object of Employee function gets created by JS Engine using global Object() constructor function (using new operator)   Like **var empProto = new Object();**   1. Since the Object() function is called using “new” operator – So the Global Object function too have the Prototype object. 2. The Object Function’s prototype object has a **dunder proto** too, which points to null. 3. This here is the point where the Prototype chaining terminates. |

## ES6 CONCEPTS

**ES6 FEATURES**

1. **VARIABLE DECLARATION LET AND CONST**
2. **IIFE V/S BLOCKS.**
3. **ARROW FUNCTIONS**
4. **DE STRUCTURING**
5. **ARRAYS**
6. **SPREAD OPERATOR**
7. **REST AND DEFAULT PARAMETER**
8. **MAPS**
9. **CLASSES AND SUBCLASSES**

### VARIABLES

|  |  |
| --- | --- |
| **CONST**   * const variables **cannot** be re-assigned * It has block scope * Const variables cannot be used before it is declared. * The const varibables are not hoisted | const toDoList = ['reading', 'walking'];  toDoList.push(“eating”); <- Valid  toDoList = []; 🡨 This gives error |
| **LET**   * let variables cannot be re-declared * It has a block scope * let variables cannot be used before it is declared. The let varibables are not hoisted. | let a = 10;  let a = 20; 🡨In-valid it cannot be redeclared  a = 30; 🡨 valid it can be re-assigned |
| “var” has function scope | “let” & “const” has block scope |
| for (var counter = 0; counter < 5; counter++) {  console.log(counter)  }  console.log(counter)); 🡨 counter is accessible outside for block and prints **5**. | for (let counter = 0; counter < 5; counter++) {  console.log(counter)  }  console.log(counter); 🡨 counter is not accessible outside the for block so it will be undefined here. |
| var counter=20;  for (var counter = 0; counter < 5; counter++) {  console.log(counter)  }  console.log(counter) 🡨 o/p = 5 | let counter=20;  for (let counter = 0; counter < 5; counter++) {  console.log(counter)  }  console.log(counter) 🡨 o/p = 20 |

### STRING CONCATENATION – TEMPLATE LITERALS

|  |  |
| --- | --- |
| For String concatenation we usually use “+”.  Now in ES6 we can use a ***back tick*** symbol and use ${} to append dynamic values. They are called “***Template Literals***” | const name = "Adams";  const yearofBirth = 1982;  function calculateAge(yearofBirth) {  return new Date().getFullYear() - yearofBirth;  }  console.log(`My name is ${name} and age is ${calculateAge(yearofBirth)}`); |

### OBJECT LITERALS

|  |  |
| --- | --- |
| To create a object literals , of the property name is same as the parameter name , we can directly the parameters as is. | |
| **OLD WAY** | **ES6 WAY** |
| function getBook(author, title) {  return {  author: author,  title: title  }  }  var book = getBook('JK', "Harry Potter");  console.log(book); | function getBook(author, title) {  return {  author,  title  }  }  var book = getBook('JK', "Harry Potter");  console.log(book); |

### ARROW FUNCTIONS

|  |  |
| --- | --- |
| 1. Arrow function is a shorter version of an anonymous function. 2. We **cannot call an arrow function** before declaring | SYNTAX  () => {statements;} |
| **FUNCTION EXPRESSION**  var myFn = function () {  console.log(“Hello”);  };  **myFn();** | **CORRESPONDING ARROW FUNCTION**  var myFn = ()=> {  console.log(“Hello”);  };  **myFn();** |
| **ARROW FUNTION WITH PARAMETERS** | |
| **ARROW FUNTION WITH ONE PARAMETERS** | **ROW FUNTION WITH 2 OR MORE PARAMETERS** |
| var showData = **data** =>{  console.log(data);  };  showData(10);  When we have one parameter parenthesis are optional so can be removed | var addData = (op1, op2) => {  console.log(op1+op2);  };  addData(1,2);  For more than one parameters parenthesis are mandatory |
| **FOR SINGLE STATEMENT BRACES ARE OPTIONAL**  var showData = **data** => console.log(data);  showData(10); | **RETURN STATEMENTS ARE OPTIONAL**  var addData = (op1, op2) => op1+op2;  **addData**(1,2); |
| **The “this” of arrow function never switches it context. It always refers to the context where it is defined** | |

### DESTRUCTURING

* Destructing helps in extracting the values from a data structure like arrays or object
* Destrructing never changes the actual array or object.

|  |  |
| --- | --- |
| **ARRAY DESTRUCTURING** | In array de-structuring the order matters.  **EXAMPLE 4:**  let numbers =[1,2,3];  **[a, ,b]=numbers;**  console.log(a,b) 🡨o/p a=1; b= 3 |
| **EXAMPLE 1**  const person = ["Adam", "25", "Programmer"];  let [name, age, profession] = person;  console.log(`${name} is ${age} year old and work as a ${profession}`);  **EXAMPLE 2 : Destructuring with REST Parameter**  let names =['Max','rob','Alex'];  let [teacher,...students] = names;  console.log(teacher, students );  **EXAMPLE 3 : Destructuring in declaration**  let [a,b,c] =[1,2,3]; |
| **EXAMPLE 5: DEFAULT VALUE**  In destructuring if the value is not available. It can also be initialized with some default value.  let names =['Max','rob'];  let [student1,student2, student3="Alex"] = names;  console.log(student1,student2,student3 );  **EXAMPLE 6: SWAPING VALUE**  let a= 10;  let b=20;  [b,a]= [a,b];  console.log(a,b); |
| **OBJECT DESTRCUTURING**  const person = {  name: "Adam",  age: "25",  profession: "Programmer"  };  let { name, age, profession } = person;  console.log(`${name} is ${age} year old and work as a ${profession}`);  let { name: n, age: a, profession: p } = person; 🡨**Alias**  console.log(`${n} is ${a} year old and work as a ${p}`); | * In object de-structuring we can give an alias name of the object properties as well and can access the properties using alias names as shown in example |
| function calculateAgeAndRetirementAge(dob) {  let age = new Date().getFullYear() - dob;  return [age, 65 - age];  }  let [age, retirementAge] = calculateAgeAndRetirementAge(1982);  console.log(`Age is ${age} and retirement age is ${retirementAge}`); | De-structuring can be used to return multiple values from a function |

### DEFAULT PARAMETERS

|  |  |  |
| --- | --- | --- |
| * The default parameter is a way to set default values for function parameters if the value is no passed in for that parameter * The default parameters should be the last argument in the function parameters. | function add(op1, op2, op3 = 30) {  console.log(op1);  console.log(op2);  console.log(op3);  }  add(10, 20); | function add(op1, op2 = 20, op3) {  console.log(op1);  console.log(op2);  console.log(op3); 🡨 this will be undefined  }  add(10, 20); |
| * JavaScript also allows the use of arrays and null as default values. | function add(op1, op2, op3 = null){ } | var arr = ["a", "b", "c"];  function add(op1, op2, op3 = arr) {  console.log(op1);  console.log(op2);  console.log(op3[2]); 🡨 this will print “**c**”  }  add(10, 20); |

### SPREAD OPERATOR

|  |  |  |
| --- | --- | --- |
| In a regular function which has multiple parameters. We call the way we are doing . There is a better way of doing in ES5 and ES6 | | function addAges(age1, age2, age3, age4) {  return age1 + age2 + age3 + age4;  }  console.log(addAges(1, 2, 3, 4)); |
| **FUNCTION CALLS** | | |
| **ES5 (using apply)** | | **ES6 : SPREAD IN FUNCTION CALLS** |
| function addAges(age1, age2, age3, age4) {  return age1 + age2 + age3 + age4;  }  let ages = [1, 2, 3, 4];  console.log(addAges.apply(null, ages)); | | function addAges(age1, age2, age3, age4) {  return age1 + age2 + age3 + age4;  }  let ages = [1, 2, 3, 4];  console.log(addAges(...ages)); |
| **ARRAY LITERALS** | | |
| **ES5** | | **ES6: SPREAD IN ARRAY LITERALS** |
| **ARRAY CONCATENATION**  var arr1 = [0, 1, 2];  var arr2 = [3, 4, 5];  arr1 = arr1.concat(arr2); 🡨 Merged Array | | **ARRAY CONCATENATION USING SPREAD**  var arr1 = [0, 1, 2];  var arr2 = [3, 4, 5];  var mergedArray = [...arr1, 10, ...arr2, 12]; 🡨 Merged Array |
| **OBJECT EXPRESSIONS** | | |
| **ES5** | **ES6: SPREAD IN OBJECT EXPRESSIONS** | |
|  | var obj1 = { foo: 'bar', x: 42 };  var obj2 = { foo: 'baz', y: 13 };  var clonedObj = { ...obj1 };  // Object { foo: "bar", x: 42 }  var mergedObj = { ...obj1, ...obj2 };  // Object { foo: "baz", x: 42, y: 13 } | |
| **SPREAD** **DEFINATION**  The Spread operator expand /spread the elements  Spread syntax allows an iterable such as   * an array expression or string to be expanded in places where zero or more arguments (for function calls) or * elements (for array literals) are expected, * or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected. | | |

### REST PARAMETER

|  |  |  |
| --- | --- | --- |
| * The Rest parameters syntax looks exactly same as spread operator, but the functionality is just opposite to it. * The rest parameter syntax allows us to represent an indefinite number of single valued arguments as an array. * With the help of a rest parameter a function can be called with any number of arguments, * ***It should be the last argument in the function.*** |  | |
| **ES5** | | **ES6** |
| function isFullAge() {  var ageArr = Array.prototype.slice.call(arguments);  ageArr.forEach(currentElement => {  console.log(2020 - currentElement >= 18);  });  }  isFullAge(1982, 1997, 2010); | | function isFullAge**(...ages**) {  ages.forEach(currentElement => {  console.log(2020 - currentElement >= 18);  });  }  isFullAge(1982, 1997, 2010); |
| function fun(...input) {  let sum = 0;  for (let i of input) {  sum += i;  }  return sum;  }  console.log(fun(1, 2)); 🡨3  console.log(fun(1, 2, 3)); 🡨6  console.log(fun(1, 2, 3, 4, 5)); 🡨15 | | // rest with function and other arguments  function fun(a,b,...c){  console.log(`${a} ${b}`); //Mukul Latiyan  console.log(c); //[ 'Lionel', 'Messi', 'Barcelona' ]  console.log(c[0]); //Lionel  console.log(c.length); //3  console.log(c.indexOf('Lionel')); //0  }  fun('Mukul','Latiyan','Lionel','Messi','Barcelona'); |

***Note: Spread operators is used in function call but Rest Parameters are used in function declaration***

### FOR OF LOOP

|  |  |
| --- | --- |
| For..og loop is new addition in ES6 for iteration | let names =['Max','rob','Alex'];  for(let name of names)  console.log(name); |

### MAPS

Earlier to store key value pair we were using object , Maps are better to store key value pairs

* In maps we can store anything as a key but in object value are limited to string type.
* Maps are iterable but object are hard to iterate

|  |  |  |  |
| --- | --- | --- | --- |
| **CREATING A MAP** | var studentMap = new Map(); | **ITERATING A MAP** | |
| **SETTING DATA IN MAP** | studentMap.set(1, "Alex"); | **Using forEach Loop** | **Using for of Loop** |
| **GETTING DATA FROM MAP** | studentMap.get(1); | **Using for of Loop**  **var studentMap = new Map();**  **studentMap.set(1, "Alex");**  **studentMap.set(2, "Mary");**  **studentMap.forEach((key, value) => console.log(key + " " + value));** | var studentMap = new Map();  studentMap.set(1, "Alex");  studentMap.set(2, "Mary");  for (let [key, value] of studentMap.entries()) console.log(key + " " + value); |
| **DELETEING FROM MAP** | studentMap.delete(1) 🡨 1 is the key |
| **CLEARING A MAP** | studentMap.clear(); |
| **CHECK KEY EXIST** | studentMap.has(1) 🡨 1 is the key |
| **GETTING ALL KEYS** | studentMap.keys(); |  |  |
| **GETTING ALL VALUES** | studentMap.values(); |  |  |

### WEAKMAP

* The keys of WeakMap are reference types , like object . It cannot be primitive type.
* Having the restriction of keeping the keys of reference types help the browser to identify which items in Weakmap is longer in use and hence can be garbage cleaned. The helps in the memory management
* The WeakMaps are not enumrable i.e. they cannot be iterated because the size of these map are variable (because the unused key/value are always gets garbage collected by the browser if they are not in use)

|  |  |
| --- | --- |
| **CREATING A WEAK MAP** | var student = new WeakMap(); 🡨 Creating a weakmap  var student1 ={rollNo:1}; 🡨 Creating keys of reference type (object)  var student2 ={rollNo:2};  student.set(student1,"Alex");🡨 Setting the values in the Map  student.set(student2,"Erica");  console.log(student.get(student1)); 🡨- Getting the value fron Map |

### SETS

* It is a data structure with no duplicate values

|  |  |  |
| --- | --- | --- |
| **CREATING A SET** | var studentNames = new Set(); | **ITERATING A SET** |
| **SETTING DATA IN SET** | studentNames.add("Alex"); | **Using for of Loop**  for(let studentName of studentNames)  console.log(studentName); |
| **GETTING DATA FROM SET** | studentNames.get(“Alex”); |
| **DELETEING FROM SET** | studentNames.delete(“Alex”) |
| **CLEARING SET** | studentNames.clear(); |
| **CHECK VALUE EXIST IN SET** | studentNames.has(“Alex”); |

### WEAKSET

* WeakSet stores the values of reference types.
* Having the restriction of keeping the values of reference types help the browser to identify which items in WeakSet is longer in use and hence can be garbage cleaned. The helps in the memory management

|  |  |
| --- | --- |
| **CREATING A WEAK SET** | var student = new WeakSet(); 🡨 Creating a weakset  var student1 = {rollNo:1,name:"Alex"};🡨 Creating object of reference type (object)  students.add(student1);🡨 Setting the values in the Set |

## ASYNCHRONOUS JAVASCRIPT

1. The Asynchronous code never block the code , this is something run behind the scene
2. Refer “Event Loop”

|  |  |
| --- | --- |
| **SAMPLE ASYNC CODE** | **CALLBACK** |
| function getDataFromService(requestData, callback) {  setTimeout(function(){  callback({ responseData: 10 });  }, 5000);  }  getDataFromService(5, function (data) {  console.log("Data From Service=" + data.responseData);  }); | var getGitHubData = function(userId) {  $.ajax({  url: "https://api.github.com/users/" + userId,  success: function(response) {  $.ajax({  url: "https://api.github.com/users/" + response.login + "/repos",  success: function(repoResponse) {  console.log(repoResponse);  },  error: function(error) {  console.log(error);  }  });  },  error: function(error) {  console.log(error);  }  });  };  getGitHubData("avishekhsinhaRepo"); |
| 1. In Async calls when the request of a async call depends on the result of previous call , we keep on nesting the calls 2. For more such calls / nesting thing becomes unmanageable. This scenario is called a **“CALLBACK HELL”** 3. ES6 comes with a solution which call solve the problem of Callback hell – The Solution has been catered using PROMISES. 4. Promises are used to deal with asynchronous JavaScript. |

### PROMISES

**WHAT IS PROMISE?**

* Promise is an object which keep track about weather a certain event has happened or not
* And if that event happened what will happen after the event has occurred.

**Here event means aync events like fetching data from server**

|  |  |
| --- | --- |
| **STATES OF PROMISE**   1. Before the event has happened, we call the state has pending 2. Once the event has happened the promise goes in resolved state. 3. if the promise is resolved , it can resolved as successful or error 4. If it is successful it’s called **FULFILLED** and **REJECTED** if it’s failed. |  |

**IMPORTANT POINTS ON PROMISE**

1. With respect to code standpoint we can **PRODUCE PROMISES AND CONSUME PROMISES**
2. When we CREATE A PROMISE, we create an instance of promise and send the result for consumption.
3. When we CONSUME PROMISES, we use callback function to accept the result send by the Promise(in Step2)
4. While consuming we have 2 callback functions
   1. one if the promise is FULFILLED called the success callback
   2. Another if promise in REJECTED called error callback

#### PRODUCING PROMISE

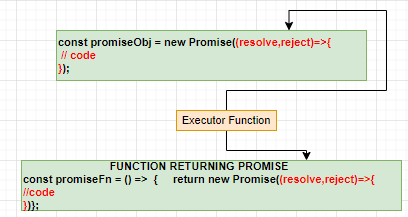
1. The promise object is created using new keyword
2. The function passed in the argument of Promise object is called Executor function

##### EXECUTOR FUNCTION

The Executor function :

* 1. Has two argument “resolve” &” reject”.
  2. In executor function we place the async code.

They are callback functions which will be called based on the Promise States (FULLFILLED OR REJECT)



##### RESOLVING AND REJECTING PROMISES

|  |  |
| --- | --- |
| **RESOLVING PROMISES** | **REJECTING PROMISES** |
| const getGitHubData = new Promise(function (resolve, reject){ **resolve**(response);  });  When the promise is FULFILLED **resolve()** callback is called | const getGitHubData = new Promise(function (resolve, reject) {  **reject**(response);  });  When the promise is REJECTED **reject()** callback is called |
| **EXAMPLE 1**  const promiseObj = new Promise((resolve,reject)=>{  resolve({id:2});  });  promiseObj.then((result)=>{  console.log(result);  }).catch(error =>{  console.log(error);  }); | **EXAMPLE 2**  const promiseFn = () => { return new Promise((resolve,reject)=>{  resolve({id:1});  })};  promiseFn().then((result)=>{  console.log(result);  }).catch(error =>{  console.log(error);  }); |

#### CONSUMING PROMISE

1. To consume promise we use then() method. Then() method returns a promise and can be chained.
2. It takes 2 arguments : a success callback and a error callback.

|  |  |  |
| --- | --- | --- |
| **WHEN PROMISE IS RESOLVED**  getGitHubData.then(function(data){ }); | **WHEN PROMISE IS REJECTED**  getGitHubData.catch(function(data){ }); | |
| **PUTTTING IT ALL TOGETHER** | | |
| var getGitHubData = new Promise(function(resolve, reject) {  **resolve**({ login: "loginId" });  });  getGitHubData.then(function(**data**) {  console.log(data.login);  }) .catch(function(error) {  console.log(error);  }); | var getGitHubData = new Promise(function(resolve, reject) {  **reject**(new Error("Error Occured"));  });  getGitHubData.then(function(data) {  console.log(data.login);  }) .catch(function(**error** ) {  console.log(error);  }); | |
| * When the promise is fulfilled we call resolve() * The result data is received in the argument of resolve * When we call the resolve function the result data is the passed to the callback function of **then**() | * When the promise is rejected we call reject() * The error data is received in the argument of reject() * When we call the reject function the error data is the passed to the callback function of **catch**() | |
| **PUTTING IT ALL TOGETHER** | | |
| **CREATING A PROMISE** | | **CONSUME THE PROMISE** |
| var makeRequest = function(url) {  var request = new XMLHttpRequest(); // Create the XHR request  // Return it as a Promise  return new Promise(function(resolve, reject) {  request.onreadystatechange = function() {  // Setup our listener to process completed requests  if (request.readyState !== 4) return;  // Only run if the request is complete  if (request.status >= 200 && request.status < 300) {  // Process the response  **resolve**(request.response); // If successful  } else {  **reject**({ status: request.status, statusText: request.statusText }); // If failed  }  };  request.open("GET", url, true); // Setup our HTTP request  request.send(); // Send the request  });  }; | | var url = “<https://api.github.com/users/avishekhsinhaRepo>”;  makeRequest(url).then(  data => {  console.log(data);  }).catch( error => {  console.log("Promise rejected."+ error.message);  }); |
| **PROMISE IMPL USING JQUERY AJAX METHOD** | | |
| var gitHubPromise = function(gitUrl) {  return $.ajax(gitUrl);  };  ***The Ajax function returns Promise so we can call “then()” and “catch()” method on that promise object*** | | gitHubPromise("https://api.github.com/users/avishekhsinha Repo") .then(function(data) {  console.log(data);  }) .catch(function(error) {  console.log(error);  }); |

### PROMISES CHAINING

|  |  |  |
| --- | --- | --- |
| const students=[  { id:1, name:'Alex' },  { id:2, name:'John' },  { id:3, name:'Rob' }  ];  const hobbies= [  { id:1, hobbies:[ 'Sports', 'Cooking'] },  { id:2, hobbies:[ 'Hiking', 'Cricket'] },  { id:3, hobbies:[ 'Tracking', 'Chess'] }  ];  const getStudent = (studentName)=> {  return new Promise((resolve,reject)=>{  const student = students.find(element => element.name == studentName);  if(!student) reject(new Error("Student Not Found"));  resolve(student.id);  });  } | | const getHobbies = (studentid)=> {  return new Promise((resolve,reject)=>{  const student = hobbies.find(element => element.id == studentid);  if(!student) reject(new Error("No Associated Hobbies"));  resolve(student.hobbies);  });  }  getStudent('Rob').then(data=> getHobbies(data).  then(hobbies => console.log(hobbies)))  .catch(err =>{ console.log(err.message)}); |
| **CONSUMING PROMISES** | **PRODUCING PROMISES USING AJAX CALL** | |
| var getGitDetailsForUser = function(userId) {  getLoginDetails(userId)  .then(function(data) { #1  let loginId = JSON.parse(data).login;  return getRepoDetails(loginId); #2  }) .then(function(data) {  console.log(data);  }) .catch(function(error) {  console.log(error);  });  };  getGitDetailsForUser("avishekhsinhaRepo");  ***Note:#1 value will become input at #2 and so on*** | **Usually producing promises usually we do via some library like JQuery**  const getRepoDetails = function(loginDetails) {  return $.ajax("https://api.github.com/users/" + loginDetails + "/repos");  };  const getLoginDetails = function(userId) {  return $.ajax("https://api.github.com/users/" + userId);  };   * getRepoDetails & getLoginDetails are Promise Object | |

### ASYNC AND AWAIT

|  |  |
| --- | --- |
| * When we chain the Promises while consuming the promises- it looks still becomes clumsy and hard to maintain for that we use **async and await** * async and await is used when we consume promises. * The function which is consuming the promises is marked with **async and the promises are marked as await.** * ***The async function always return a PROMISE object*** and executes in background without blocking the regular code flow. We need to use await on in async function * For example while consuming the promises .First #1 promise will execute till that time #2 promise will wait . Once #1 is completed #2 will execute. | |
| **IMPLEMENTING ASYNC AND AWAIT**  **The above example in promise chaining can be also achieved by async and await** | |
| const getStudentAsync = async (name) => {  const studentId = await getStudent(name);  const studentHobbies = await getHobbies(studentId);  console.log(studentHobbies);  }  getStudentAsync('Rob'); | **IF ASYNC function returns a value , the returned value can be intercepted using then()**  const getStudentAsync = async (name) => {  const studentId = await getStudent(name);  const studentHobbies = await getHobbies(studentId);  return (studentHobbies);  }  getStudentAsync('Rob').then(hobbies => console.log(hobbies)); |

#### HOSTING

1. Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution. This means that no matter where functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local.

|  |  |  |
| --- | --- | --- |
| * In the JS script code since compilation step occurs before it actually get executed. In the compilation step the code is scanned and variable are identified and set into the global /function scope * Due to the compilation step it does actually matter where the variables are declared, it stills holds good. * The variable declarations are hoisted on the top and assigned the value “**undefined**” | ACTUAL CODE | HOSTED CODE BY JS ENGINE |
| a=10;  console.log(b);  b=20  c++;  **var a;**  **var b;**  **var c;** | **var a;**  **var b;**  **var c;**  a=10;  console.log(b); 🡪undefined  b=20  c++; |
| **FUNCTION** | | |
| We call the function expression before declaration. **ITS VALID** | fn();  function fn() {  } | function fn() {  }  fn(); |
| **FUNCTION EXPRESSION** | | |
| We cannot call the function expression before declaration. **ITS INVALID** | fn();  var fn = function(){  } |  |
| For the given code, the highlighted variable prints “**undefined**”. Because the variable in the function is also hoisted inside the function as shown. | var greetings = "Hello!";  function greetMe() {  console.log(greetings);  var greetings = "Hi";  }  greetMe(); | var greetings;  greetings = "Hello!";  function greetMe() {  var greetings;  console.log(greetings);  greetings = "Hi";  }  greetMe(); |

## CLASSES IN JAVASCRIPT

* Classes introduced in ES6. They are just syntactical sugar on the old way of creating object and implementing inheritance using function constructor.
* They also act as a blueprint for all the objects we create using the class.

|  |  |
| --- | --- |
| **CREATING OBJECT** | |
| **ES5 – USING FUNCTION CONSTRUCTOR** | **ES6 – USING CLASS** |
| var Person = function(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  };  Person.prototype.getFullName = function() {  return this.firstName + " " + this.lastName;  };  let person = new Person("Mike", "Clark");  console.log(person); | class Person {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  getFullName() {  return this.firstName + " " + this.lastName;  }  }  let person = new Person("Mike", "Clark");  console.log(person); |
| **Person.prototype 🡨** Here “**Person**” is Function Constructor | **Person.prototype 🡨** Here “**Person**” is a class |
| * When we create a method in a class it actaully creates a Person’s class prototype. * **The classes in JavaScript are not hosited** , means we can use class only after its declaration unlke functuon constructor. * ***We can only add methods to classes not properties*** * ***The ES6 class body are always execusted in strict mode*** | |

### SYNTAX

|  |  |
| --- | --- |
| **SAMPLE CLASS** | **DETAILS** |
| class Person {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  get fullName() { #2  return this.firstName + " " + this.lastName;  }  set fullName(name) { **#3**  let parts = name.split(" ");  this.firstName = parts[0]; this.lastName = parts[1];  }  static isSamePerson(person1, person2) { #4  return (  person1.firstName == person2.firstName &&  person2.lastName == person2.lastName  );  }  }+  let mike = new Person("Mike", "Doe");  let john = new Person("John", "Clark");  console.log(mike.fullName);  john.fullName = "Dan Wills"; 🡨 **Call to setter**  console.log(john.fullName); 🡨 **Call to getter**  console.log(Person.isSamePerson(mike, john)); 🡨 **Calling a static function** | 1. The class can be declared using “**class**” keyword 2. The class can have constructor (parameterized / non-parameterized)   **GETTERS & SETTERS**   1. Use “**get**” keyword to create a getter.#2- is getter of an “fullName” property. 2. Use “**set**” keyword to create a getter.#3- is setter of an “fullName” property.   **STATIC FUNCTION**   1. #4 – We use “**static’** keyword to declare a static function. 2. Static functions are never a part of an object , but its part of class, so static functions are called using class name(**#11**) 3. The static function are not inherited   **Note :**   1. ***When we set the value of a property it implicitly calls its setter*** 2. ***When we access a value of a property it implicitly calls its getter.*** |

### INHERITANCE

|  |  |
| --- | --- |
| **ES5** | **ES6** |
| function Shape() {  this.x = 0;  this.y = 0;  }  Shape.prototype.move = function(x, y) {  this.x += x;  this.y += y;  console.info('Shape moved.');  };  function Rectangle() {  **Shape.call(this);** // call super constructor.  }  **Rectangle.prototype = Object.create(Shape.prototype); #1**  **Rectangle.prototype.constructor = Rectangle; #2**  **var rect = new Rectangle();**  console.log('Is rect an instance of Rectangle?', rect instanceof Rectangle); 🡪 true  console.log('Is rect an instance of Shape?', rect instanceof Shape); 🡪 true  rect.move(1, 1); 🡪 Outputs, 'Shape moved.' | class Person {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  }  class Programmer extends Person {  constructor(firstName, lastName, skill) {  super(firstName, lastName);  this.skill = skill;  }  }  let mike = new Programmer("Mike", "Wilson", "Java");  console.log(mike);   * Person is the parent class and Programmer is the child class which inherits it. * super() 🡪 Calls the constructor of parent class |
| **DETAILS ON INHERITANCE IN ES5: USING FUNCTION CONSTRUCTOR**   1. Shape is the super class of Rectangle Object 2. **move**() has been created in Shape object (Shape prototype) so that it can be shared by all the object created from Shape Function Constructor. 3. **new Rectangle(); 🡪** This calls the constructor of Rectangle which has a call to constructor of super class [**Shape.call(this);**]. “this” object here will be object of Rectangle type. 4. **#1🡪**subclass extends superclass 5. **#2 🡪**if you don't set **Object.prototype.constructor** to Rectangle, it will take prototype.constructor of Shape (parent). To avoid that, we set the prototype.constructor to Rectangle (child). | |

### POLYMORPHISM

|  |  |
| --- | --- |
| **ES5** | **ES6** |
|  | class Person {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  hobby() {  return "Music";  }  }  class Programmer extends Person {  constructor(firstName, lastName, skill) {  super(firstName, lastName);  this.skill = skill;  }  hobby() {  return "Games";  }  }  let person = new Person("Dan", "Wilson");  console.log(person.hobby());  person = new Programmer("Mike", "Wilson", "Java");  console.log(person.hobby()); |

## HTML 5 WEB WORKER

**WHAT DO WEB WORKERS DO?**

1. This help us in executing multiple threads in parallel
2. The web worker consumes lot of resources. So must use when it’s truly needed.
3. The threads are kernel level thread which runs on client machine

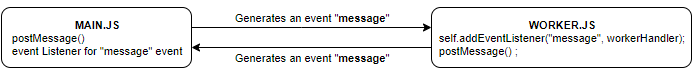
**TYPES OF WEB WORKERS?**

1. Dedicated Web workers
2. Shared Web worker

### DEDICATED WEB WORKERS

1. Dedicated web worker are created and managed by main thread.
2. The Dedicated web worker is forced to quit when the main thread terminates.

**CREATING DEDICATED WEB WORKERS**



**HOW MAIN JS AND WORKER JS COMMUNICATE?**

1. The main JS pass the values from main.js to worker JS via postMessage() function argument
2. Along with passing the value ,the main thread triggers an event “message” as well
3. On the worker we need to implement the event listener for “message” event to receive the vale
4. Once the worker finished it task the workers then triggers a “message” event and pass the value to main thread function using postMessage() function argument
5. The main thread need to implement the event handler for “message” event which receives the result.

|  |  |  |
| --- | --- | --- |
| **HTML** | <input type="text" name="txt" id="txt" />  <input type="button" onclick="getResult()" /> | **STEPS TO CREATE WEB WORKER**   1. #1 🡪Check whether the web worker is supported 2. #2 🡪 Creates an object of Worker with worker js file as argument. 3. The code in the worker file runs in different thread asynchronously. 4. #3 🡪 Success Event handler of “**message**” event 5. #4 🡪 Error Event handler of “**message**” event 6. #5 🡪 postMessage() passes the value to web worker 7. #6 🡪 Event listener of “message” event |
| **MAIN JS** | function getResult() {  if (Worker != undefined) { #1  var txtValue = document.getElementById("txt").value;  var worker = new Worker("worker.js"); #2  worker.onmessage = successHandler; #3  worker.onerror = errorHandler; #4  worker.postMessage(txtValue); #5  }  }  function successHandler(e) {  console.log(**e.data**);  }  function errorHandler(e) {  console.log(**e.data**);  } |
| **WORKER JS** | self.addEventListener("message", workerHandler); #6  function workerHandler **e.data** (e) {  postMessage(+ " " + "World!"); #7  } |

**HANDLING ERRORS - CUSTOM ERRORS**

|  |  |
| --- | --- |
| **MAIN JS** | **WORKER JS** |
| function add() {  if (Worker != undefined) {  var txtValue1 = document.getElementById("number1").value;  var txtValue2 = document.getElementById("number2").value;  var worker = new Worker("worker.js");  worker.onmessage = successHandler;  worker.onerror = errorHandler;  var requestObj = {  num1: txtValue1,  num2: txtValue2  };  worker.postMessage(**requestObj**);  }  }  function successHandler(e) {  if (e.data.errorMsg) {  console.log("Invalid Input");  } else {  console.log("Sum=" + e.data);  }  }  function errorHandler(e) {  console.log(e.message + " in " + e.filename);  } | self.addEventListener("message", add);  function add(e) {  if (isNaN(e.data.num1) || isNaN(e.data.num2)) {  var errorObj = {  errorMsg: "Not a Number"  };  postMessage(errorObj);  } else {  var result = parseInt(e.data.num1) + parseInt(e.data.num2);  postMessage(result);  }  } |
| **HTML**  <input type="text" name="txt1" id="number1" />  <input type="text" name="txt2" id="number2" />  <input type="button" onclick="add()" /> |

**IMPORTING SCRIPTS**

* We can able to import js files in the worker files as well using importScript(“<file\_path>”)
* Below example shows the way to import a JS file which in the same directory , where worker is present.

|  |  |
| --- | --- |
| **WORKER FILE** | **FILE THAT HAS TO BE IMPORTED** |
| **importScripts("workerHelper.js");**  self.addEventListener("message", add);  function add(e) {  if (isNaN(e.data.num1) || isNaN(e.data.num2)) {  var errorObj = {  errorMsg: "Not a Number"  };  postMessage(errorObj);  } else {  var result = addNumbers(e.data.num1, e.data.num2);  postMessage(result);  }  } | function addNumbers(num1, num2) {  return parseInt(num1) + parseInt(num2);  } |
| This helps in centralizing the logic in a common file and the web worker can import the file and use the common code. |

**TERMINATING THE WEBWORKER**

* Since the workers are OS kernel level threads so they consume lot of resources. We should terminate the web worker when it’s not needed. There are 2 ways we can terminate the web worker

|  |  |
| --- | --- |
| 1. **MAIN THREAD TERMINATES THE WEB WORKER**   Main thread can terminates the webworker using terminate() function  **<webwork\_object>.terminate();**  **MAIN.JS**  var worker;  function getResult() {  if (Worker != undefined) {  var txtValue = document.getElementById("txt").value;  worker = new Worker("worker.js");  worker.onmessage = successHandler;  worker.onerror = errorHandler;  worker.postMessage(txtValue);  }  }  function successHandler(e) {  console.log(**e.data**);  **worker.terminate();**  }  function errorHandler(e) {  console.log(**e.data**);  **worker.terminate();**  } | 1. **WEB WORKER TERMINATES ITSELF**   Web worker thread can terminates itself the webworker using close() function  **self.close()**  **WORKER.JS**  self.addEventListener("message", workerHandler);  function workerHandler **e.data** (e) {  postMessage(+ " " + "World!");  **self.close();**  } |

### SHARED WEB WORKERS

|  |  |
| --- | --- |
| **CREATING SHARED WEB WORKER**   1. When the main thread triggers the shared webworker Unlike dedicated worker the shared worked keep on executing even if the main thread terminates 2. Multiple main threads can communicate with shared web worker, but the main thread will share the same instance of Shared worker. 3. Note – Any error in shared web worker is never reported back to the main thread, as they run independently. | For example let’s consider we want to create an application which will increase the count by 1 of every button click |

# TYPESCRIPT

**WHAT IS TYPESCIPT?** - It’s a open source project by Microscoft. It’s a superset of JavaScript

**WHY TYPESCRIPT? -**Typescript help us to overcome the problem in Javascript. For example .

**ISSUES WITH JAVASCRIPT:-**

1. Lack type checking. For example,

var a =10;

a=”Hello” //allowed in JS

1. Issue with Function

**Function Defination** : function add(a,b){ return a+b; }

**Function call** :

* 1. add(1) 🡪 O/P = NaN // Allowed in JS
  2. add(1,2,3) 🡪 O/P = 3 // Third parameter has been ignored

1. **Object Modification –** Lets say we have an object

var person ={ 'firstName': 'John','lastName':'Doe'} **-** This javascript object can be modified at any point of time . We don’t have a way to control

1. Adding new propery is fairly simple = person.age=25

Final object will be : {firstName: "John", lastName: "Doe", age: 25}

These all limitations or quirks can be handled in Typescript.

|  |  |
| --- | --- |
| **HOW TYPESCRIPT WORKS?**   1. We write a code in Typescript 2. Typescript compiler compiles the Typescript to Native Javascipt code. Typescript compiler used Node.js as its compiler 3. The Advantage we get here as follows As the TS compiler compiled to native JS so it is understandable by all the browsers |  |

### TYPESCRIPT DEVELOPMENT ENVIRONMENT SET UP

1. Install Editor Visual Studio code
2. Install the latest stable version **NODE**

**WHAT IS NODE?** Earlier the runtime environment for Javascript is browser. Node provides the runtime environment for Javascript outside the browser.

|  |  |
| --- | --- |
| * Create a folder where you want to create your typescript file * Now to install typescript we need to install it using npm(node package manager). | |
| **CREATING A TS PROJECT**  Step 1 : Create a directory and generate a package.json file : **npm init --yes** | |
| **INSTALLING TS DEPENDENCIES** | **npm install typescript ts-node-dev**  Here **–g** Means we want to install typescript globally.  Otherwise TS will be installed in the folder where we are in |
| **COMPILE TS FILE** | **tsc** <type\_script\_file\_name> . e.g. tsc hello-world – ***the “ts” extension is optional*** |
| **RUN COMPILED JS** | node <js\_file\_name>.js e.g. node hello-world.js |
| **TO WATCH THE CHANGES** | 1. Install the dev dependency : npm install --save-dev **lite-server 🡨** lite server is a development server 2. **Update the start script in package.json**   **"start":"lite-server"** |

|  |  |  |
| --- | --- | --- |
| **Typescript code** | **Compiled JS Code** | When the typescript code is compiled the compiler takes off the types. So, the type declaration in TS comes in picture at compile time only. |
| var a :number;  var b:string;  var c:boolean;  a=10;  b='hello';  c=true; | var a;  var b;  var c;  a = 10;  b = 'hello';  c = true; |
| var a :number; a=10; a='Hello';  In the above declaration a is a number type. We are still assigning the string to it. TS compiler will complain about it at compile time , but it will still generate the JS output file as below . Because it a valid JS syntax  var a;  a = 10;  a = 'Hello'; |

### TYPESCRIPT TYPES

* NUMBER
* STRING
* BOOLEAN
* UNDEFINED
* NULL
* TUPLE
* UNION
* LITERAL
* ANY
* ENUMS

|  |  |  |
| --- | --- | --- |
| **Datatype** | **Declaration** | **Details** |
| Number | var a:number; | “a” can store only number values |
| Boolean | var a:boolean; | “a” can store only boolean values |
| String | var a:string; | “a” can store only string values |
| Undefined | var a:undefined; | “a” can store only undefined |
| Null | var a:null; | “a” can store only null |
| Tuple | var a:[number,Boolean];  **it’s a fixed length and fixed type datatype** | Example :  const user : {  name:string;  age:number;  role:[string,number];  } ={  name:'John',  age:30,  role:['admin',1]  } |
| a can store an arrays of 2 elements . e.g.  var myTuple:[number,boolean];  myTuple=[1,true]; |
| Array | var a:number[]; | a is an array which can store numbers |
| Object type (Nested object) | Let's say you have this JavaScript object:  const product = {  id: 'abc1',  price: 12.99,  tags: ['great-offer', 'hot-and-new'],  details: {  title: 'Red Carpet',  description: 'A great carpet - almost brand-new!'  }  } | This would be the type of such an object:  {  id: string;  price: number;  tags: string[],  details: {  title: string;  description: string;  }  } |

#### ENUM

|  |  |
| --- | --- |
| enum Roles {ADMIN, AUTHOR, READ\_ONLY};  const user = {  name:'Max',  age:'30',  role: Roles.ADMIN  }  if(user.role == Roles.ADMIN){  console.log("Role is Admin");  } |  |

#### FUNCTION TYPE

|  |  |
| --- | --- |
| function add(number1:number, number2:number){  return number1 + number2;  }  let addFn : Function = add;  console.log(addFn(1,3)); |  |

### TS FUNCTIONS

|  |  |
| --- | --- |
| JS WAY | TS WAY |
| function addNumber(a,b){ return a+b; }   1. console.log(addNumber("foo",1)); O/P – **foo1** 2. console.log(addNumber(1,2,3)); O/P - **3**   Here the JS function has no type check for the function parameter passed  #2 – We can pass more parameters to the function. JS function simply ignores the extra parameters. | function addNumber(a :number,b:number){  return a+b;  }   1. console.log(addNumber(2,1)); 2. console.log(addNumber(2));   in TS we have a type check for function parameter types and number of parameters passed to it.#2 will give an error by TS compiler |
| **PASSING VARIABLE PARAMETERS** | |
| function addNumber(a :number, b:number,c?){  return a+b+c;  } | Here c is a optional parameter  console.log(addNumber(2,1,3)); // Valid TS function call  console.log(addNumber(2,1)); // Valid TS function call |

### IMPLICIT TYPING (TYPE INFERENCE)

|  |  |
| --- | --- |
| **EXPLICIT TYPING** | **IMPLICIT TYPING** |
| var a=10:number;  a='hello' 🡨TS compiler gives an error as it’s a string | var a=10  a='hello' 🡨 TS compiler gives an error as it’s a string . Here TS compiler understands that “a” is meant for number type values. |

##### TYPESCRIPT FUNCTION



### TYPESCRIPT CLASSES AND FUNCTION

Member Variable can have both implicit and Explicit Typing as highlighted below.

|  |  |
| --- | --- |
| **CLASS DECALARATION** | |
| class Person{  firstName : string;  lastName :string;  **constructor**(firstName , lastName){  this.firstName = firstName;  this.lastName = lastName  }  getFullName(){  return this.firstName +" "+ this.lastName;  }  } | * firstName and lastName are member variables * getFullName() is a method of the class. * constructor(..) is a constructor of Person class   Note :   * ***In TS classes cannot have overloaded constructor and method***.   **CREATING OBJECT FROM CLASS**  var aPerson = new Person("John", "Doe"); console.log(aPerson.getFullName()); |

#### PRIVATE & PUBLIC ACCESS MODIFIERS

|  |  |
| --- | --- |
| class Employee {  private name:string;  private department:string[]= [];  constructor(name:string){  this.name = name;  }  addDepartment(departmentName:string){  this.department.push(departmentName);  }  printEmployeeInfo(){  console.log(this)  }  }  const mike = new Employee("mike");  mike.addDepartment('Technology');  mike.addDepartment('Sales');  mike.prprintEmployeeInfo(); | 1. The member variables are by default public variables and can be accessed from anywhere. 2. TS has access modifies like private, public and protected. 3. All private variable call be accessed only withing the class. |

###### SHORTHAND FOR INITIALIZATION

|  |  |
| --- | --- |
| class Employee {  private department:string[]= [];  constructor(private name:string, private id:string){ }  addDepartment(departmentName:string){  this.department.push(departmentName);  }  printEmployeeInfo(){  console.log(this)  }  }  const mike = new Employee("mike", 'EMP001');  mike.addDepartment('Technology');  mike.addDepartment('Sales');  mike.printEmployeeInfo(); | * Intilaization of properties can be done in constructor using a shorthand * In this way the parameter passed in constructor and the class property has to be same. |

#### READ ONLY MODIFIERS

|  |  |
| --- | --- |
| class Employee {  private department:string[]= [];  constructor(private name:string, private **readonly** id:string){ }  } | * The readonly modifier allow the property to initialize just once. * There after it cannot be modiflied even in the method inside the class. |

### INHERITANCE

|  |  |
| --- | --- |
| **EXTENDING A CLASS(INHERITANCE)** | |
| class Human {  constructor(private firstName : string, private lastName : string){}  }  class Student extends Human{  constructor(firstName : string, lastName : string,  private id :string,private department :string){  **super(firstName,lastName);**  }  }  let john = new Student('John','Doe','ST001','Engineering');  console.log(john); | * Extends keyword is used to inherit the classes * Super keyword is used to call the implementation of super classes * TypeScript does not support multilevel inheritance. |

#### PROTECTED MODIFIERS

* Protected properties can be accessed within the class and the class extending that base class.

### GETTERS AND SETTERS IN CLASSES

|  |  |
| --- | --- |
| class Account{  private \_balance=0;  set balance(balance:number){ 🡨 Setter  this.\_balance = balance;  }  get balance(){🡨 getter  return this.\_balance;  }  } | const account = new Account();  account.balance= 100; 🡨 Calling Setter  console.log("Balance:" , account.balance); 🡨 Calling Getter |

### STATIC PROPERTIES AND METHOD

|  |  |
| --- | --- |
| class Product{  static currentYear:number =2021;  constructor(private productId:string, private productName:string, private manuFacturingYear:number){}    static isCurrentYearProduct(product:Product){  return Product.currentYear == product.manuFacturingYear;  }  }  const pr1 = new Product("P001","Product1",2021);  console.log(Product.isCurrentYearProduct(pr1)); | * Static properties and method are used to create global constants and utility function * They can be accessed using Class name. * Static member cannot be accessed using “this” keyword |

### ABSTRACT CLASSES

|  |  |
| --- | --- |
| abstract class Shape{  **abstract draw():void;**  }  class Circle extends Shape{  constructor(private radius:number){  super();  }  draw() {  console.log("Draw Circle");  }  }  const circle = new Circle(1);  circle.draw(); |  |

### SINGLETON AND PRIVATE CONSTRUCTORS

|  |  |
| --- | --- |
| class SimpleSingleton{  private static instance: SimpleSingleton;  private constructor(){}  static getInstance(){  if(SimpleSingleton.instance){  return this.instance;  }  this.instance = new SimpleSingleton();  return this.instance;  }  }  const i1 = SimpleSingleton.getInstance();  const i2 = SimpleSingleton.getInstance();  console.log(i1,i2,i1===i2); |  |

### INTERFACES

|  |  |
| --- | --- |
| **INTERFACES** | |
| interface Person {  firstName: string;  lastName: string;  getFullName(): string;  }  ***The Person interface reference can hold the object of classes which implements Person interface*** | class Teacher implements Person {  firstName: string;  lastName: string;  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  getFullName(): string {  return this.firstName + " " + this.lastName;  }  }  let mike: Person = new Teacher("Mike", "Holding");  console.log(mike.getFullName()); |
| * Interfaces also help is creating own custom types * Here **confimPassword** is an optional property   This declares an object of type User  user = { userName: "John", password: "Hopkin" }; | interface User {  userName: string;  password: string;  confimPassword?: string;  }  let user: User; 🡨 variable declared to custom type (User)  user = { userName: "John", password: "Hopkin" };  console.log(user); |

### MODULES IN TYPESCRIPT

1. We can modularize the typescript source code into multiple files
2. We have to export the interfaces and classes from one file and import it in another file(using file name)

|  |  |
| --- | --- |
| **EXPORT INTERFACE** | **IMPORTED IN A CLASS** |
| export interface Human {  firstName:string;  lastName :string;  getFullName():string;  } | import {Human} from './typescript-interfaces';  class Person implements Human{  firstName : string;  lastName : string;  constructor(firstName:string, lastName:string){  this.firstName = firstName;  this.lastName = lastName;  }  getFullName() : string {  return "Hello !"+" "+this.firstName + " "+ this.lastName;  }  } |
| When we import a file in another file typescript executes the imported file  **OUTPUT**  From Person  Hello World ! John Doe | export class Person{ }  **console.log("From Person");**  import {Person} from './person';  class Programmer extends Person{  getFullName() : string {  return "Hello World !"+" "+this.firstName + " "+ this.lastName;  }  }  var aPerson: Person = new Programmer("John","Doe");  console.log(aPerson.getFullName()); |

### TYPESCRIPT COMPILER ARGUMENTS

#### WATCH MODE

|  |  |
| --- | --- |
| **DESCRIPTION** | **COMMAND** |
| To create a compiled JS file with desired output file name | tsc .\typescript-compiler.ts --out compiledFile.js |
| To watch the changes in the typescript file   * It constantly monitor the changes of TS file * It picks the changes, compiles it and generate the output | tsc .\typescript-compiler.ts --out compiledFile.js –watch |
| The downside of this approach is that – the above command will only watch one single file for the changes.  The compiler arguments can be configured in a configuration file (tsconfig.json).It helps to ease the compilation process | |

#### CREATE TSCONFIG FILE

1. **The tsconfig.file is basically a configuration file to manage the TS projects**
2. Run the command : **tsc –init**

##### TS CONFIG FILE CONFIGRATIONS

###### WATCHING THE CHANGES IN MULTIPLE FILES

* Creating the tsconfig gives the flexibility to compile all the TS file in one go
* This also enable the watch mode for all files

|  |  |
| --- | --- |
| Step 1: Generate the tsconfig.json file. | **tsc --init** |
| Step 2: To compile all the files [tsc] | **tsc** |
| Step 3 : To watch all the files | **tsc --w** |

###### EXCLUDING AND INCLUDING FILES

|  |  |
| --- | --- |
| {  "compilerOptions": {  /\* Basic Options \*/  },  "exclude": [  /\*Path of file to be excluded from compilation \*/  "analytics.ts",  "\*.dev.ts",  "\*\*/\*.dev.ts",  "node\_modules"  ],  "include": [  /\*Include the file and file path to include  ]  } | * Tsconfig.json also helps in configuring the exclusion and inclusion of files and folders in the compilation * We can use folder path and regex to define the paths |

###### SETTING UP COMPILATION TARGET

|  |  |  |
| --- | --- | --- |
| {  "compilerOptions": {  "target": "es5"  /\* Specify ECMAScript target version: 'ES3' (default), 'ES5', 'ES2015', 'ES2016', 'ES2017', 'ES2018', 'ES2019' or 'ESNEXT'. \*/  }  } | | |
| const employee ={  name:'Dan',  age:30  }  let empCopy = employee;  console.log(empCopy); | When Target is es5. The ts file will be compiled to es5 javascript  "use strict";  var employee = {  name: 'Dan',  age: 30  };  var empCopy = employee;  console.log(empCopy); | When Target is es5. The ts file will be compiled to es6 javascript  "use strict";  const employee = {  name: 'Dan',  age: 30  };  let empCopy = employee;  console.log(empCopy); |

###### SETTING UP OUTPUT FOLDER

To set the location(output folder) of compiled script.



### CREATING A TYPESCRIPT PROJECT

To create a typescript project we need to follow below steps

1. **CREATE A TS.CONFIG FILE**
2. **CREATE A NODE PROJECT**

# CREATE A NODE PROJECT

|  |  |
| --- | --- |
| 1. Run a command “**npm -init**” to create a **package.json** file. 2. Press “Enter” to create npm project with default values.   **SAMPLE PACKAGE.JSON FILE**  **{**  **"name": "typescript-basics",**  **"version": "1.0.0",**  **"description": "",**  **"main": "index.js",**  **"scripts": {**  **"test": "echo \"Error: no test specified\" && exit 1"**  **},**  **"author": "",**  **"license": "ISC"**  **}**   1. **main** 🡪 is the starting point of the application. 2. So in TS we can create a index.ts file which when compiled creates a index.js file, which ultimately becomes the starting point of the application. |  |

#### COMPILING AND EXECUTING TS FILES

To understand the compilation and execution, lets create 2 files

|  |  |
| --- | --- |
| person.ts | Index.ts |
| **export** class Person {  firstName: string;  lastName: string;  constructor(firstName: string, lastName: string) {  this.firstName = firstName;  this.lastName = lastName;  }  getFullName(): string {  return this.firstName + " " + this.lastName;  }  } | i**mport { Person } from "./person";**  let per = new Person("Mike", "Clark");  console.log(per.getFullName()); |
| 1. Person class has been exported from person .ts which in turn imported in index.ts |
| **COMPILATION** :  Command For compilation : **tsc**  This will create the compiled JS file in output folder configured in tsconfig.json(outputFolder) in our example |
| 1. **EXECUTION**   **Command for execution : node ./outputFolder/index.js** |
| This seems to be a lengthy process we can simply the compilation and execution step using package.json configuration   1. Add “start“ property and assign the chained command , which will compile and execute the index.js file   TO COMPILE & RUN USE COMMAND : **npm start** | **{**  **"name": "typescript-basics",**  **"version": "1.0.0",**  **"description": "",**  **"main": "index.js",**  **"scripts": {**  **"start": "tsc && node outputFolder/index.js",**  **"test": "echo \"Error: no test specified\" && exit 1"**  **},**  **"author": "",**  **"license": "ISC"**  **}** |

#### MANAGING DEPENDENCIES

* One of the using a node project is that its ease the process of managing dependencies . Below are the steps how node help us in managing dependent libraries. Let’s say we want to use “lodash” library(this libabry has lot of utility functions)

|  |  |
| --- | --- |
| **INSTALLING LODASH LIBRARY**  **Generic Command to install a library**  **Npm install <Library\_name> --save** | **FOR LODASH**  **npm install lodash --save** |
| This command will do the following   1. Add the lodash library in the node\_module 2. Add a dependency in the package.json file. |  |
| **INSTALLING LODASH TYPE DEFINITION**  **Generic Command to install the type definition library**  **Npm install @type/<Library\_name> --save-dev** | **FOR LODASH**  **npm install @types/lodash --save-dev**  **Note ,**   * Installing type definition give suggestions in the IDE and type checking functionality for the depended libraries. * The type definitions are not the TS reimplementation of lodash JS library , it just contains the type definition. |
| **USING LIBRARY IN TS FILES** | import \* as **\_lodash** from "lodash";  let arr = [5, 4, 3];  console.log**(\_lodash.reverse(arr));** |

#### SAMPLE TYPESCRIPT PROJECT

# ANGULAR 10

## OVERVIEW

|  |  |
| --- | --- |
|  | 1. Angular has component-based approach 2. Each component should be sufficient on its own 3. The components are registered using some tag/ selector |

## SETTING UP THE DEVELOPMENT ENVIRONMENT

To set up the development environment for Angular JS, we need to install

1. **Node JS** – This provides the runtime environment for the angular application. It has bunch of utilities that help us to build and run the angular application
2. **Microsoft Visual Studio** – Editor
3. **Angular CLI** – The Angular CLI is a tool to initialize, develop, scaffold, and maintain Angular applications.

|  |  |
| --- | --- |
| **TO VALIDATE THAT NODE IS INSTALLED. USE THE BELOW COMMAND TO VALIDATE.**   1. node –v: This gives the version of the node installed 2. npm –v : This gives the version of the package manager installed. | TO VALIDATE THAT ANGULAR CLI IS INSTALLED. USE THE BELOW COMMAND TO VALIDATE.   1. To install the Angular CLI: **npm install -g @angular/cli** 2. To check completion and version: **ng –version** |

## CREATING ANGULAR 6 PROJECT

The Angular CLI helps us to create a bare bone Angular project for us

**CLI COMMANDS**

|  |  |
| --- | --- |
| 1. To create a new Angular Project: **ng new <*project\_name*>.**    1. This will create an Angular Project and install the dependencies needed    2. It will be an npm project. 2. To run the Angular App : **ng serve** 3. Access the application from browser : <http://localhost:4200/>   **EXAMPLE**  **ng new my-first-project**  **cd my-first-project**  **ng serve**  ng serve will start a development server locally  Note : ng serve or npm start are 100% same command |  |

**WHAT ANGULAR CLI DO?**

1. Angular CLI has created a bare bone angular project
2. It will create a single component (app – component) - which will act as a root component from all other custom components.
3. The root component will act as a placeholder for the other entire custom component.

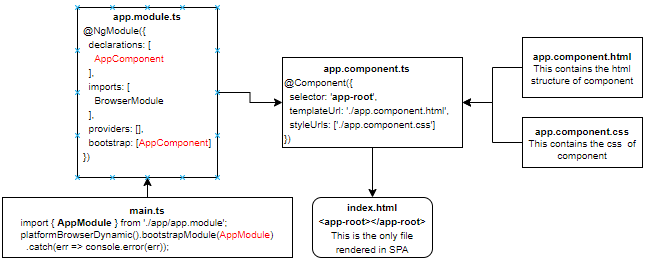
## FOLDER STRUCTURE

|  |  |
| --- | --- |
|  | 1. **src** : This folder will have our custom/source code will go. 2. **node\_module** : This will have all the dependencies installed /downloaded need to run the Angular App 3. **package.json** : This has the list of dependencies(similar to pom.xml) 4. **e2e** : Need for **end to end** testing of Angular 6 application. |

## ROOT COMPONENT IN DETAILS

|  |  |  |
| --- | --- | --- |
|  | Any component in angular is consist of below file need for rendering :   1. Front End view(HTML markup) – This resides in the app.componet.html 2. The backend logic – This resides in app.component.ts file, which is a Typescript file/code. 3. The component can be used in an html file (index.html) using a tag. 4. Style : app.component.css 5. app.component.spec.ts : This file will all the test cases to unit test the component | |
| TS FILE [**app.component.ts**]  import { Component } from '@angular/core';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent {  title = 'first-project';  } | USING THE COMPONENT (IN INDEX.HTML)  <html lang="en">  <body>  **<app-root></app-root>**  </body>  </html> |

## HOW ANGULAR APPLICATION IS BOOTSTRAPED



1. The app module is the root module - which is bootstrapped by **main.ts** file
2. The app.module file has declaration of all the components which has to be known to angular before the app component is bootstrapped (bootstrap: [**AppComponent**])
3. The Module - bundles all the components.
   1. The components that must be bundled is declared in “declarations” array in @NgModule. This declaration happens automatically when we create a component using angular cli.
4. app component is the root component of complete project

## CSS IN ANGULAR PROJECTS

|  |  |
| --- | --- |
| * style.css = This css file is meant for global styliing * For each component we create – there will eb dedicated css also created for that component only * The CSS added in component specific css (e.g. app.component.css) is applied to that component only. |  |

### INSTALLING BOOTSTRAP CSS USING NPM

|  |  |
| --- | --- |
| INSTALL THE BOOTSTRAP CSS USING NPM: **npm install --save bootstrap@3**  ADDING THE BOOTSTRAP MODULE TO PROJECT   * GO TO angular.json file on the project * Add the bootstrap css path in the styles array |  |
| CSS can also be added using import statement in style.css file | EXAMPLE : To include bulma css library   1. npm install bulma 2. Go to style.css and write the import statement as   **@import 'bulma/css/bulma.css';** |

### ADDING IMAGE TO PROJECT

|  |  |
| --- | --- |
| To add image to the project. The images can be kept in the asset folder of the project  REFERING IMAGES IN HTML : **<img src="assets/tree.jpeg">** |  |

## CREATING THE FIRST CUSTOM COMPONENT

|  |  |
| --- | --- |
| 1. The custom component will be the child of root component. 2. The component can be used in the root component html(app.component.html) using its selector |  |
| **CLI Command t**o create a new component :   * ***ng generate component <component-name>*** * ***ng g c <component-name>***   **CREATING A COMPONENT WITHOUT TEST FILE**   * ***ng generate component <component-name> --spec false***   **CREATING A COMPONENT IN A FOLDER**   * ***ng generate component <folderpath>/<component-name>*** * This has created 4 files for the new component * It has updated the app.module.ts * The new component has been created inside app folder. | |  |
| As stated – The module bundles all the components so we need to declare all the components that has to be bundles into a module  So - Once the component is created – it will add the declaration in the root module   1. Import the component using import statement (Note – don’t had “.ts” extension) 2. Declare the component in the declaration section of Module 3. The complete object is then passed to NgModule decorator. | | import { **HelloComponent** } from './hello/hello.component';  @NgModule({  declarations: [ AppComponent,  **HelloComponent**  ],  imports: [ BrowserModule ],  providers: [],  bootstrap: [AppComponent]  }) |
| **HOW TO USE THE COMPONENT**   1. Get the selector name from hello-world.component.ts file 2. Go to root component html(app.component.html) 3. Use the selector using the selector in **app.component.html**   ***<app-hello-world></app-hello-world>***   1. Everytime we use the tag – we are creating an instance of the corresponding TS clas | |  |

**IMPORTANT NOTES:-**

1. The component TS file is the entry point of any angular component. This bundles all the files of the component.
2. The TS file has been annotated with @Component to declare it as an Angular Component. Below are the details of @Component Annotation properties.

|  |  |
| --- | --- |
| selector | This is the name by which a component is registered with |
| templateUrl | This is the file where the component’s markup resides |
| styleUrl | It’s an array , where the CSS can be listed for a component |

### COMPONENT SELECTORS

#### ATTRIBUTE SELECTORS

|  |  |
| --- | --- |
| **COMPONENT** | **HTML** |
| @Component({  selector: '[app-server]',  templateUrl: './server.component.html',  styleUrls: ['./server.component.css']  })  export class ServerComponent implements OnInit { ..} | <div app-server>  </div> |

#### CLASS SELECTORS

|  |  |
| --- | --- |
| **COMPONENT** | **HTML** |
| @Component({  selector: '.app-server',  templateUrl: './server.component.html',  styleUrls: ['./server.component.css']  })  export class ServerComponent implements OnInit { } | <div class=”app-server”>  </div> |

## DATA BINDING

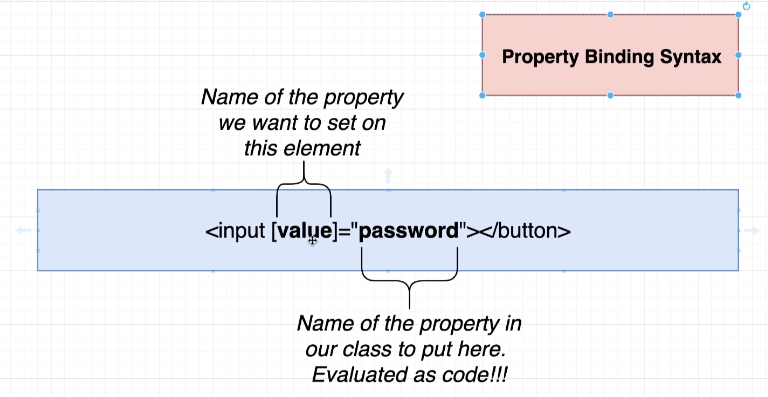
|  |  |
| --- | --- |
|  | **The data binding can happen either of the ways**  1. Typescript to HTML using String interpolation and Property binding  2. ***HTML to Typescript*** – Event Binding like click event  3. Two-way data binding can be achieved using ngModel. |

### STRING INTERPOLATION - ONE WAY DATA BINDING

Let’s create a new a component – “server” component to demonstrate data binding.

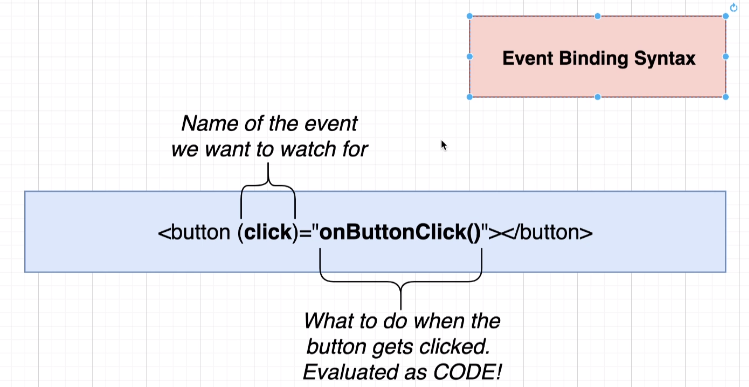
|  |  |
| --- | --- |
| **COMPONENTS TS FILE – server.component.ts** | **COMPONENT HTML – server.component.html** |
| @Component({  selector: "app-server",  templateUrl: "./server.component.html",  styleUrls: ["./server.component.css"]  })  export class ServerComponent implements OnInit {  serverId: number = 10;  serverStatus: string = "offline";  constructor() {}  getServerStatus() {  return this.serverStatus;  }  ngOnInit() {}  } | <p>Server {{ serverId }} is {{ getServerStatus() }}</p>**.**   1. The string interpolation operator can enclose any such variable which is a string or can be easily type converted to string link integer 2. It can also enclose methods |

### PROPERTY BINDING - ONE WAY DATA BINDING



|  |  |
| --- | --- |
| @Component({  selector: "app-server",  templateUrl: "./server.component.html",  styleUrls: ["./server.component.css"]  })  export class ServerComponent implements OnInit {  **allowedToAddSevers**: boolean = false;  constructor() {  setTimeout(() => {  this.allowedToAddSevers = true;  }, 2000);  }  ngOnInit() {}  } | <button class="btn btn-primary" [disabled]="!allowedToAddSevers">  Add Server  </button>  This one-way binding is with a HTML element property like **disabled** |

### EVENT HANDLING



|  |  |
| --- | --- |
| @Component({  selector: "app-server",  templateUrl: "./server.component.html",  styleUrls: ["./server.component.css"]  })  export class ServerComponent implements OnInit {  serverCreationStatus = "No Server Added !";  constructor() { }  serverAdded() {  this.serverCreationStatus = "Server Added";  } | ngOnInit() {}  } |
| <button class="btn btn-primary" **(click)="serverAdded()">**Add Server</button> |
| **PASSING VALUE IN EVENT BINDING** | |
| import { Component, OnInit } from "@angular/core";  @Component({  selector: "app-server",  templateUrl: "./server.component.html",  styleUrls: ["./server.component.css"]  })  export class ServerComponent implements OnInit {  serverName: string = "";  onCreateServer(event) {  this.serverName = event.target.value;  }  ngOnInit() {}  } | <input type="text" (input)="**onCreateServer($event)"** class="form-control" />   * **Here we are binding input event** * **$event 🡪 is the reserved word** |

### TWO WAY DATA BINDING [ngModel Directive]

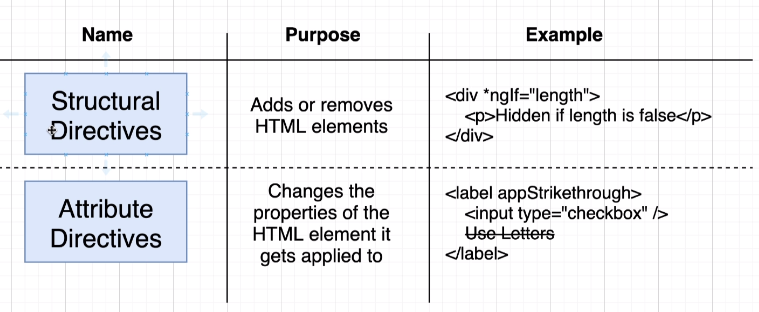
1. To enable the ngModel directive. We need to add the FormsModule to the imports[] array in the AppModule.
2. Add the import from @angular/forms in the app.module.ts file: **import { FormsModule } from '@angular/forms';**

|  |  |  |
| --- | --- | --- |
| import { BrowserModule } from "@angular/platform-browser";  import { NgModule } from "@angular/core";  import { FormsModule } from "@angular/forms";  import { AppComponent } from "./app.component";  import { ServerComponent } from "./server/server.component";  import { ServersComponent } from "./servers/servers.component";  @NgModule({  declarations: [AppComponent, ServerComponent, ServersComponent],  imports: [BrowserModule, FormsModule],  providers: [],  bootstrap: [AppComponent]  })  export class AppModule {} | @Component({  selector: "app-server",  templateUrl: "./server.component.html",  styleUrls: ["./server.component.css"]  })  export class ServerComponent implements OnInit {  serverName: string = "";  constructor() {}  ngOnInit() {}  } | <input type="text" [(ngModel)]="serverName" /> |

## ANGULAR DIRECTIVES

* **THE DIRECTIVES ARE USED TO *MODIFY THE STRUCTURE OR PROPERTIES OF HTML ELEMENT***
* All structural directives start with \* e.g. \*ngIf,\*ngFor etc. otherwise called as attribute directive

|  |  |
| --- | --- |
| **ATTRIBUTE DIRECTIVES** | **STRUCTURAL DIRECTIVES** |
| It looks like normal HTML attribute | Looks like normal HTML attribute but has leading \* |
| Only affect /change the element they are added to like styling | Affect the whole area in the DOM(elemnts are added or removed) |



|  |  |
| --- | --- |
| Note – Two structural directives cannot be placed in the same element together | <div \*ngFor="let num in numbers" \*ngIf="num % 2 ==0">  </div |

### PRE-DEFINED DIRECTIVES

#### STURUCTURAL DIRECTIVES

|  |  |  |
| --- | --- | --- |
| **\*ngIf directive** | | |
| @Component({  selector: "app-server",  templateUrl: "./server.component.html",  styleUrls: ["./server.component.css"]  })  export class ServerComponent implements OnInit {  serverCreationStatus = "No Server Added !";  isServerAdded: boolean = false;  serverAdded() {  this.serverCreationStatus = "Server Added";  this.isServerAdded = true;  }  ngOnInit() {}  } | <button  class="btn btn-primary"  [disabled]="!allowedToAddSevers"  (click)="serverAdded()"  >  Add Server  </button>  <p \*ngIf="isServerAdded">{{ serverCreationStatus }}</p> | |
| **If else directive** | | |
| If we want to add else condition | <p \*ngIf="isServerAdded; **else noServer**">{{ serverCreationStatus }}</p>  <ng-template #noServer>  <p>No Server Added</p>  </ng-template> | |
| **\*ngFor** | | |
| @Component({  selector: "app-server",  templateUrl: "./server.component.html",  styleUrls: ["./server.component.css"]  })  export class ServerComponent implements OnInit {  servers = [];  server = "";  constructor() {}  ngOnInit() {}  onCreateServer() {  console.log(this.server);  this.servers.push(this.server);  }  } | | <input type="text" class="form-control" [(ngModel)]="server" />  <button class="btn btn-primary" (click)="onCreateServer()">  Add servers  </button>  <ul>  <li \*ngFor="let server of servers; let serverNumber = index">  {{ serverNumber }} - {{ server }}  </li>  </ul> |

### ATTRIBUTE DIRECTIVES

#### ngClass and ngStyle

|  |  |
| --- | --- |
| **APP.TS** |  |
| export class AppComponent {  listOfNumbers: number[] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  }  **APP.HTML**  <div class="container">  <div class="row">  <div class="list-group">  <a href="#" class="list-group-item"  **\*ngFor="let num of listOfNumbers"**  **[ngClass]="num%2==0?'active':''"**  **[ngStyle]="{color:num%2!=0 ?'red':''}">{{num}}</a>**  </div>  </div>  </div> |
| * ngStyle expect a JavaScript object |

#### ng-container

#### NgSwitch

|  |  |
| --- | --- |
| **TS FILE**  import { Component, OnInit, Input } from "@angular/core";  @Component({  selector: "app-comment",  templateUrl: "./comment.component.html",  styleUrls: ["./comment.component.css"]  })  export class CommentComponent implements OnInit {  value = 5;  constructor() {}  ngOnInit() {}  } | **HTML**  <div [ngSwitch]="value">  <p \*ngSwitchCase="1">Value is 1</p>  <p \*ngSwitchCase="5">Value is 5</p>  <p \*ngSwitchDefault>default</p>  </div> |

### CUSTOM DIRECTIVES

#### CUSTOM ATTRIBUTE DIRECTIVES

|  |  |  |
| --- | --- | --- |
| **CREATING A DIRECTIVE** | | 1. **ng g d simpledirective** 2. **ng generate directive simpledirective** |
| **APP MODULE** | | **GENERATED DIRECTIVE TS** |
| import { BrowserModule } from "@angular/platform-browser";  import { NgModule } from "@angular/core";  import { FormsModule } from "@angular/forms";  import { AppComponent } from "./app.component";  import { SimpledirectiveDirective } from './simpledirective.directive';  @NgModule({  declarations: [AppComponent, SimpledirectiveDirective],  imports: [BrowserModule, FormsModule],  providers: [],  bootstrap: [AppComponent]  })  export class AppModule {} | | import { Directive } from "@angular/core";  **@Directive({**  **selector: "[appSimpledirective]"**  **})**  export class SimpledirectiveDirective {  constructor() {}  } |
| 1. The newly created directive must be declared at the App Module level |
| **CREATING AND USING AN ATTRIBUTE DIRECTIVE**   * In this example we will create a directive which will change the background color of the elment on hovering it. * The directive always manipulates the HTML element , so to create a custom directive we need to get hold of the element itself to manipulate | | |
| **TS FILE**  import { Directive, ElementRef, OnInit, Renderer2 } from "@angular/core";  @Directive({  selector: "[appSimpledirective]"  })  export class SimpledirectiveDirective implements OnInit {  constructor(private elementRef: ElementRef, private renderer: Renderer2) {}  ngOnInit(): void {  this.renderer.setStyle( this.elementRef.nativeElement,"background-color", "red" );  }  } | **HTML**  <p **appSimpledirective**>This is from Directive</p> | |
| * ElementRef 🡪 This helps in getting hold of the element on which the directive is applied.   This can be also achieved using  **this.elementRef.nativeElement.style.backgroundColor = "green";**   * The renderer object have multiple methods for DOM manipulation (<https://angular.io/api/core/Renderer2>) | |

**EXAMPLE 2**

|  |  |
| --- | --- |
| **DIRECTIVE (CUSTOM)** | **TEMPLATE CODE** |
| import { Directive, HostListener, HostBinding } from '@angular/core';  @Directive({  selector: '[appDropdown]'  })  export class DropdownDirective {  constructor() { }  @HostBinding("class.open") isOpen: boolean = false;  **@HostListener('click') t**oggleDropDown() {  this.isOpen = !this.isOpen;  }  } | <ul class="nav navbar-nav navbar-right">  <li class="dropdown" **appDropdown**>  <a class="dropdown-toggle" data-toggle="dropdown">Manage  <span class="caret"></span></a>  <ul class="dropdown-menu">  <li><a>Save Data</a></li>  <li><a>Fetch Data</a></li>  </ul>  </li>  </ul> |
| * In the above example – We are using a bootstrap drop down. To open the drop down we need to add “open” class in <li> * Step 1: We need to bind a variable with the class (open) – using Hostbinding * Step 2: Bind a function which will manipulate the variable on a binded event – using HostBinding | |

##### HOSTLISTENER AND HOSTBINDING DECORATOR – EXAMPLE 2

**CUSTOM PROPERTY BINDING IN DIRECTIVES**

|  |
| --- |
| <p appSimpledirective [mousehovercolor]="'yellow'" [mouseleavecolor]="'green'"> This is from Directive</p>  **Here we are handing the color (unlike hardcoded in previous example) from the teamplate code itself.** |
| import { Directive, ElementRef, OnInit, Renderer2, HostListener, Input } from "@angular/core";  @Directive({  selector: "[appSimpledirective]"  })  export class SimpledirectiveDirective implements OnInit {  constructor(private elementRef: ElementRef, private renderer: Renderer2) {}  **@Input() mousehovercolor = "";**  **@Input() mouseleavecolor = "";**  ngOnInit(): void {  this.renderer.setStyle( this.elementRef.nativeElement,"background-color","red" );  }  @HostListener("mouseover") mouseover() {  this.renderer.setStyle( this.elementRef.nativeElement,"background-color", **this.mousehovercolor** );  }  @HostListener("mouseleave") mouseleave() {  this.renderer.setStyle(this.elementRef.nativeElement,"background-color", **this.mouseleavecolor** );  }  } |

#### CUSTOM STRUCTURAL DIRECTIVES

The structural directive get transforms into ng-template as shown below

|  |
| --- |
| <div \*ngIf="condition"> This is ngIf</div> **🡪** <ng-template [ngIf]="condition"> This is ngIf</ng-template> |
| In general - Ng-template is basically display / hide the element based on the condition applied to it |

*Let create a structural directive that behaves opposite of “ngIf”*

|  |  |
| --- | --- |
| **TS FILE(DIRECTIVE)** | **HTML** |
| import { Directive, TemplateRef, ViewContainerRef, Input } from "@angular/core";  @Directive({  selector: "[appOppositeif]"  })  export class OppositeifDirective {  @Input() set appOppositeif(condition: boolean) {  if (!condition) this.vcRef.createEmbeddedView(this.tempRef);  else this.vcRef.clear();  }  constructor(  private tempRef: TemplateRef<any>,  private vcRef: ViewContainerRef  ) {}  } | <div **\*appOppositeif="false"**> This is appOppositeif </div> |
| 1. To manipulate the element we mark setter of the structural directive 2. We inject TemplateRef because , ultimately all the structural directives are transformed into <ng-template> |

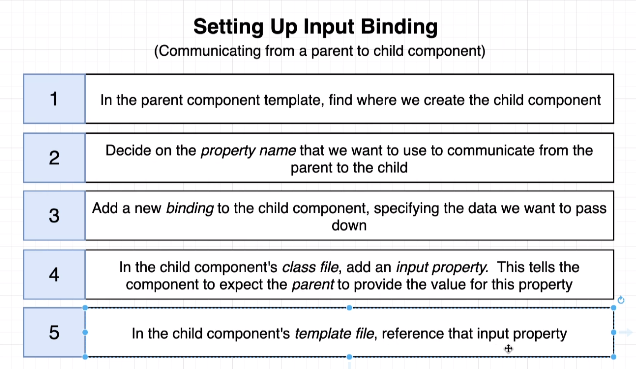
## LOCAL REFEREENCE IN TEMPLATE

* The local reference can be applied to any HTML element
* Once it is applied to the HTML element it will hold the reference o f that HTML element
* The scope of the local variable is limited to the template (HTML) where it applied. It can be passed to TS file from the template only .The TS file property cannot access to local reference unlike the two way binding.
* Once we get hold of the HTML element – so we can get or set the values of that the HTML element using local reference. It can be an alternative of two-way data binding – as shown below

|  |  |
| --- | --- |
| **STUDENT REGISTRATION HTML** | <div class="row">  <div class="form-group">  <label for="firstName">First Name</label>  <input type="text" class="form-control" id="firstName" #firstName>  </div>  <div class="form-group">  <label for="lastName">Last Name</label>  <input type="text" class="form-control" id="lastName" #lastName>  </div>  <button class="btn btn-primary" (click)="regStudent(firstName,lastName)">Register</button>  </div> |
| **STUDENT REGISTRATION TS** | import { Component, OnInit, EventEmitter, Output } from '@angular/core';  @Component({  selector: 'app-student-registration',  templateUrl: './student-registration.component.html',  styleUrls: ['./student-registration.component.css']  })  export class StudentRegistrationComponent implements OnInit {  @Output() registerStudent = new EventEmitter<{ firstName: string, lastName: string }>();  constructor() { }  ngOnInit() { }  regStudent(firstName: HTMLInputElement, lastName: HTMLInputElement) {  this.registerStudent.emit({  firstName: firstName.value,  lastName: lastName.value  });  firstName.value = "";  lastName.value = "";  } } |

## COMMUNICATION BETWEEN COMPONENTS

### PARENT TO CHILD COMMUNICATION - PROPERTY BINDING AND @Input DECORATOR



#### EXAMPLE 1

|  |  |
| --- | --- |
| **COMPONENT’S TS FILE** | **HTML FILE WHERE THE COMPONENT IS USED** |
| import { Component, OnInit, Input } from '@angular/core';  @Component({  selector: 'app-date',  templateUrl: './date.component.html',  styleUrls: ['./date.component.css']  })  export class DateComponent implements OnInit {  user :any;  @Input("name") userName:string;  constructor() { }  ngOnInit() {  this.user={  'userName':this.userName,  'firstName':'John',  'lastName':'Doe',  'address' :'New Jersey',  'phones':[  '9650758731',  '9650758732'  ]  };  }  } | **<app-date name="johnDoe"></app-date>**   1. The data “name” has been passed from the view (html) to the component. 2. **@Input annotation** is used to pass the data and populate the member variable. 3. **Input** class must be imported from angular/core module. 4. **The important point here is that we are communicating between components using property binding** |

#### EXAMPLE 2

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | | We will have 2 components   1. **Comments Component(Parent)**   This component will have a form to add comments   1. **Comment Component (Child)**   This will receive the value from the form and display the commets |
| **COMMENTS COMPONENT** | | | |
| **COMMENTS TS**  import { Component, OnInit } from "@angular/core";  @Component({  selector: "app-comments",  templateUrl: "./comments.component.html",  styleUrls: ["./comments.component.css"]  })  export class CommentsComponent implements OnInit {  comments: string[] = [];  comment: string = "";  constructor() {}  ngOnInit() {}  **addComment**() {  this.comments.push(this.comment);  console.log(this.comments);  this.comment = "";  }  } | | **COMMENTS HTML**  <div class="container">  <div class="row">  <input type="text" class="form-element" [(ngModel)]="comment" />  <button class="btn btn-primary" (click)="addComment()">Add Comment</button>  </div>  <div class="row" \*ngFor="let comment of comments">  <app-comment [comment]="comment"></app-comment>  </div>  </div> | |
| **COMMENT COMPONENT** | | | |
| **COMMENT TS**  import { Component, OnInit, Input } from "@angular/core";  @Component({  selector: "app-comment",  templateUrl: "./comment.component.html",  styleUrls: ["./comment.component.css"]  })  export class CommentComponent implements OnInit {  @Input("comment") comment = "";  constructor() {}  ngOnInit() {}  } | | **COMMENT HTML**  <p>{{ comment }}</p>  Note :   1. Here the communication is happening from Parent to Child component. | |

#### EXAMPLE 3

|  |  |  |  |
| --- | --- | --- | --- |
| **OUTPUT** | **FILE STRUCTURE** | | **COMPONENT STRUCTURE** |
|  |  | |  |
| **Blog.ts**  export class Blog {  constructor(private blogTitle, private blogContent) { }  } | | | app.component.html |
| <div class="container">  **<app-bloglist></app-bloglist>**</div> |
| **Bloglist component** | | | |
| **Bloglist.component.ts** | | **Bloglist.component.html** | |
| import { Component, OnInit } from '@angular/core';  import { Blog } from '../Blog';  @Component({  selector: 'app-bloglist',  templateUrl: './bloglist.component.html',  styleUrls: ['./bloglist.component.css']  })  export class BloglistComponent implements OnInit {  blogList: Blog[] = [];  constructor() { }  ngOnInit() {  this.blogList.push(new Blog("Blog Title 1", "Blog Content 1"));  this.blogList.push(new Blog("Blog Title 2", "Blog Content 2"));  this.blogList.push(new Blog("Blog Title 3", "Blog Content 3"));  this.blogList.push(new Blog("Blog Title 4", "Blog Content 4"));  this.blogList.push(new Blog("Blog Title 5", "Blog Content 5"));  this.blogList.push(new Blog("Blog Title 6", "Blog Content 6"));  }  } | | <app-blog [blogList]=blogList></app-blog>   1. We are passing an object ; List of blog objects | |
| **Blog component** | | | |
| **Blog.component.ts** | | **Blog.component.html** | |
| import { Component, OnInit, Input } from '@angular/core';  import { Blog } from '../Blog';  @Component({  selector: 'app-blog',  templateUrl: './blog.component.html',  styleUrls: ['./blog.component.css']  })  export class BlogComponent implements OnInit {  @Input() blogList: Blog[] = [];  constructor() { }  ngOnInit() {  }  } | | <div class="row" \*ngFor="let blog of blogList">  <div class="panel panel-default">  <div class="panel-body">  <div class="row">  {{blog.blogTitle}}  </div>  <div class="row">  {{blog.blogContent}}  </div>  </div>  </div> | |

## CHILD TO PARENT COMMUNICATION USING VIEWCHILD AND VIEWCHILDEN

* The @ViewChild and @ViewChildren decorator help in the communication betwwen child component to parent component.
* **This can used in conjunction with “local Reference” OR “Component Name” itself**

### CHILD TO PARENT COMMUNICATION USING EVENTS - @Output

|  |  |
| --- | --- |
| * App component has 2 child components. * The student-registration component has a form to Add Student * Student-list component list all the added students * The app component has student array – where new student object is pushed when added and hence student-list component list all students |  |

**HOW COMPONENTS COMMUNICATE?**

1. The student-registartion component send the newly added student by emiting a custom event (registerStudent)
2. The App component has captures the event and add it to student array
3. The student list component then shows all the added srudent using data binding
4. However – the communication among the components can also be manage by angular services as well.

|  |  |
| --- | --- |
| **APP COMPONENT HTML** | <div class="container"> <div class="row">  **<app-student-registration (registerStudent)=addStudent($event)></app-student-registration>**  </div>  <hr/>  <div class="row"><ul class="list-group">  **<app-student-list \*ngFor="let student of students" [student]=student></app-student-list>**  </ul> </div></div> |
| **APP COMPONENT TS** | export class AppComponent {  title = 'datacomm';  **students = []; 🡸*Maintains a student array shared by both registration and listing components***  addStudent(studentData: { firstName: string, lastName: string }) {  this.students.push({  firstName: studentData.firstName,  lastName: studentData.lastName  });  }  } |
| **STUDENT REGISTRATION HTML** | <div class="row">  <div class="form-group">  <label for="firstName">First Name</label>  <input type="text" class="form-control" id="firstName" [(ngModel)]="firstName">  </div>  <div class="form-group">  <label for="lastName">Last Name</label>  <input type="text" class="form-control" id="lastName" [(ngModel)]="lastName">  </div>  <button class="btn btn-primary" **(click)="regStudent()">**Register</button>  </div> |
| **STUDENT REGISTRATION TS** | import { Component, OnInit, EventEmitter, Output } from '@angular/core';  @Component({  selector: 'app-student-registration',  templateUrl: './student-registration.component.html',  styleUrls: ['./student-registration.component.css']  })  export class StudentRegistrationComponent implements OnInit {  **@Output() registerStudent = new EventEmitter<{ firstName: string, lastName: string }>();**  firstName: string;  lastName: string;  constructor() { }  ngOnInit() { }  regStudent() {  this.registerStudent.emit({  firstName: this.firstName,  lastName: this.lastName  });  this.firstName = "";  this.lastName = "";  }  } |
| **STUDENT LIST HTML** | <li class="list-group-item">{{student.firstName}} , {{student.lastName}}</li> |
| **STUDENT LIST TS** | import { Component, OnInit, Input } from '@angular/core';  @Component({  selector: 'app-student-list',  templateUrl: './student-list.component.html',  styleUrls: ['./student-list.component.css']  })  export class StudentListComponent implements OnInit {  constructor() { }  **@Input() student:** { firstName: string, lastName: string };  ngOnInit() { }  } |

### @VIEWCHILD

#### ACCESSING TEMPLATE HTML USING LOCAL REFERENCE

|  |  |  |
| --- | --- | --- |
|  | * @ViewChild decorator is helps in getting access to the component’s template HTML in the component’s TS File. * Lets say , we have a requirement to focus on the first name field on page load * The ViewChild decorator has an ability to get hold of component’s HTML * ViewChild uses “local reference” to get hold of the HTML element . * ***Note - The HTML reference will be ready and can be accessed in the ngAfterViewInit() life cycle hook*** | |
| **STUDENT REGISTRATION COMPONENT TS** | | **STUDENT REGISTRATION COMPONENT HTML** |
| import { Component, OnInit, EventEmitter, Output, ViewChild, ElementRef, AfterViewInit } from '@angular/core';  @Component({  selector: 'app-student-registration',  templateUrl: './student-registration.component.html',  styleUrls: ['./student-registration.component.css']  })  export class StudentRegistrationComponent implements OnInit, AfterViewInit {  @Output() registerStudent = new EventEmitter<{ firstName: string, lastName: string }>();  @ViewChild("firstName", { static: true }) firstName: ElementRef;  constructor() { }  ngOnInit() { }  ngAfterViewInit(): void {  this.firstName.nativeElement.focus(); **🡨 Focus on the field**  }  regStudent(firstName: HTMLInputElement, lastName: HTMLInputElement) {  this.registerStudent.emit({  firstName: firstName.value,  lastName: lastName.value  });  firstName.value = "";  lastName.value = "";  }  } | | <div class="row">  <div class="form-group">  <label for="firstName">First Name</label>  <input type="text" class="form-control" id="firstName" #firstName>  </div>  <div class="form-group">  <label for="lastName">Last Name</label>  <input type="text" class="form-control" id="lastName" #lastName>  </div>  <button class="btn btn-primary" (click)="regStudent(firstName,lastName)">Register</button>  </div> |

In the above example we are using local reference to access the HTML element of compoent’s template using @ViewChild decorator. This decorator can also be used to access the child element in the template as well

#### ACCESSING CHILD ELEMENT USING VIEW CHILD

|  |  |  |
| --- | --- | --- |
| **CHILD TS** | **PARENT HTML** | |
| export class DateComponent implements OnInit {  constructor() { }  today: Date = new Date();  ngOnInit() {  }  } | **<app-date></app-date>**  <p>User Name : {{uname}}</p>   * <app-date> is of type DateComponent which can be accessed using @ViewChild decorator. * This cannot be accesses before **ngAfterViewInit()** life cycle hook of the component | |
| **PARENT TS** | | * If we have more than on <app-date> child component – The @ViewChild can able to access only the first element. * To sccess all the <app-date> child element we need to use another decorator **@ViewChildren** |
| export class ChildcompComponent implements OnInit, **AfterViewInit** {  @Input() uname: string = "";  **@ViewChild(DateComponent, { static: true }) dateComponent: DateComponent;**  ngOnInit() { }  **ngAfterViewInit(): void {**  **setInterval(() => {**  **this.dateComponent.today = new Date();**  **}, 1000);**  **}}** | |

### @VIEWCHILDREN

|  |  |  |
| --- | --- | --- |
| **@ViewChildren** | This decorator will allow the parent to select the all instances of the child component | |
|  | In this application     1. App component has two child component student-registration and student-listing 2. The app component has 2 buttons to select and deselect the students in the list 3. The background (color changes to green on click of “select All” and “Deselect all ” will remove the background color. 4. @ViewChildren decorator helps in manipulating the child component (student-listing) from App Component. 5. App component is using local reference to access the child element | |
| **APP HTML** | | |
| <div class="container">  <div class="row">  <app-student-registration (registerStudent)=addStudent($event)></app-student-registration>  </div>  <hr/>  <div class="row">  <div class="col-md-1">  <button class="btn btn-success btn-xs" \*ngIf="students.length >0" (click)="selectAllStudent()">Select All</button>  </div>  <div class="col-md-1">  <button class="btn btn-info btn-xs" \*ngIf="students.length >0" (click)="deSelectAllStudent()">DeSelect All</button>  </div>  </div>  <hr/>  <div class="row">  <ul class="list-group">  <app-student-list \*ngFor="let student of students" [student]=student #studentList></app-student-list> 🡨 **This local reference will be used in @ViewChildren() decorator**  </ul>  </div>  </div> | | |
| **APP TS**  import { Component, ViewChildren, QueryList } from '@angular/core';  import { StudentListComponent } from './student-list/student-list.component';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent {  **@ViewChildren('studentList') studentList: QueryList<StudentListComponent> 🡨 This will give the list of all StudentListComponent(Child Component)**  title = 'datacomm';  students = [];  addStudent(studentData: { firstName: string, lastName: string }) {  this.students.push({  firstName: studentData.firstName,  lastName: studentData.lastName  });  }  selectAllStudent() {  this.studentList.forEach(element => { 🡨 **Iterating the list**  element.selectStudent(); **🡨 Calling the function in child component(StudentListComponent)**  });  }  deSelectAllStudent() {  this.studentList.forEach(element => { 🡨 **Iterating the list**  element.deSelectStudent(); **🡨 Calling the function in child component(StudentListComponent)**  });  }  } | | |
| **STUDENT LIST TS** | | **STUDENT LIST HTML** |
| import { Component, OnInit, Input } from '@angular/core';  @Component({  selector: 'app-student-list',  templateUrl: './student-list.component.html',  styleUrls: ['./student-list.component.css']  })  export class StudentListComponent implements OnInit {  selectionClass: boolean = false;  constructor() { }  @Input() student: { firstName: string, lastName: string };  ngOnInit() {  }  selectStudent() {  this.selectionClass = true;  }  deSelectStudent() {  this.selectionClass = false;  }  } | | <li class="list-group-item"  **[ngClass]="selectionClass ? 'list-group-item-success':''">**  {{student.firstName}} , {{student.lastName}} </li> |

#### ACCESSING ALL CHILD ELEMENT USING @VIEWCHILD

|  |  |  |
| --- | --- | --- |
| **CHILD TS** | **PARENT HTML** | 1. @ViewChildren decorators has a list of all child instance embedded into the HTMl template 2. As its QueryList – so we need to loop though the elements to access the child element. |
| export class DateComponent implements OnInit {  constructor() { }  today: Date = new Date();  ngOnInit() {  } } | <app-date></app-date>  <app-date></app-date>  <p>User Name : {{uname}}</p> |
| **PARENT TS**  export class ChildcompComponent implements OnInit, AfterViewInit {  @Input() uname: string = "";  @ViewChildren(DateComponent) dateComponents: QueryList<DateComponent>;  ngOnInit() {  }  ngAfterViewInit(): void {  this.dateComponents.forEach(element => {  setInterval(() => {  element.today = new Date();  }, 1000);  });  }  } | |

## VIEW ENCAPSULATION

|  |  |
| --- | --- |
| **COMPONENT** | * As a defult behavior the style in the CSS of a component is applied only that component only. * But if we want to override theis default behavior and apply the css globally. |
| import { Component, OnInit, ViewEncapsulation } from "@angular/core";  @Component({  selector: "app-blogs",  templateUrl: "./blogs.component.html",  styleUrls: ["./blogs.component.css"],  encapsulation: ViewEncapsulation.None  })  export class BlogsComponent implements OnInit {  constructor() {}  ngOnInit() {}  }  **CSS**  p { color: red; } |

|  |  |
| --- | --- |
| **ViewEncapsulation.None** | The CSS applied to the component will be applied globally |
| **ViewEncapsulation.Emulated** | Default Behavior |
| **ViewEncapsulation.Native** | Emulated behavior will be applied in those browsers which supports it otherwise applied globally |
| **ViewEncapsulation.ShadowDom** |  |

## CONTENT PROJECTION

### CONTENT PROJECTION – ngContent

|  |  |  |
| --- | --- | --- |
| **Headline Component** | When we write an HTML between the component’s selector. Angualar ignores it and pull the HTML in its own template (HTML)  When we want angular the HTML between the component’s selector we can use **<ng-content>** | |
| <app-headline>  <h1>Student Registration</h1>  </app-headline> |
| **HEADLINE COMPONENT HTML** | **APP COMPONENT HTML** | **This kind of HTML can be used to create container components. For example, headline component can have any HTML element into it.** |
| <div class="row">  <ng-content></ng-content>  </div> | <app-headline>  <h1> Student Registration</h1>  </app-headline> |

|  |  |
| --- | --- |
|  | * The content projection can be leveraged to create templates * For example, lets say we need to create a template which has fixed header and footer but different body content * We can use <ng-content> to create such templates. It acts a placeholder for content |

### MULTI SLOT CONTENT PROJECTION

1. The content can be selected too to be projected in a specifc location. The selection can be done ***class , attribute selector or id selector***

#### CONTENT PROJECTION – CLASS SELECTOR

|  |  |
| --- | --- |
| **APP HTML** | **CARD COMPONENT HTML** |
| <div class="container">  **<app-template1>**  <div class="row">  <h1>Page Heading</h1>  <hr/>  </div>  <h2 class="articleHeading">Article heading</h2>  <p class="articleContent">Article Content</p>  <div class="row">  <hr/>  <p>Page footer  </p>  </div>  **</app-template1>**  </div> | <div class="panel panel-default">  <div class="panel-heading">  **<ng-content select=".articleHeading"></ng-content>**  </div>  <div class="panel-body">  **<ng-content select=".articleContent"></ng-content>**  </div>  </div> |

#### CONTENT PROJECTION – ATTRIBUTE SELECTOR

|  |  |
| --- | --- |
| **TEMPLATE HTML** | The ng content can also select a specific HTML content based on the selector |
| <div class="container">  <div class="row">  <ng-content select="[header]"> </ng-content>  </div>  <div class="row">  <div class="col-sm-3">  <ng-content select="[leftNav]"> </ng-content>  </div>  <div class="col-sm-1"> </div>  <div class="col-sm-8">  <ng-content select="[body]"> </ng-content>  </div>  </div>  <div class="row">  <hr/>  <ng-content select="[footer]"> </ng-content>  </div>  </div> |
| **APP HTML** | |
| **<app-template>**  <app-headline header>  <h1>Student Registration</h1>  <hr>  </app-headline>  <div class="row" leftNav>  <app-student-registration (registerStudent)=addStudent($event)></app-student-registration>  </div>  <div class="row" body>  <div class="row">  <div class="col-sm-1">  <button class="btn btn-success btn-xs" \*ngIf="students.length >0" (click)="selectAllStudent()">Select All</button>  </div>  <div class="col-sm-1">  <button class="btn btn-info btn-xs" \*ngIf="students.length >0" (click)="deSelectAllStudent()">DeSelect All</button>  </div>  </div>  <hr/>  <div class="row">  <ul class="list-group">  <app-student-list \*ngFor="let student of students" [student]=student #studentList></app-student-list>  </ul>  </div>  </div>  <div class="row" footer> Footer </div>  **</app-template>** | |

### CONTENT PROJECTION - @ContentChild

* The **@ContentChild** decorator can get hold of child of the projected content.

|  |  |  |
| --- | --- | --- |
| **APP COMPONENT** | **CARD COMPONENT** | |
| <div class="container">  <app-card>  <div class="row">  <h1>Page Heading</h1>  <hr/>  </div>  <h2 class="articleHeading">Article heading</h2>  <p class="articleContent">  **<app-date></app-date>**  <span>Article Content</span>  <div class="row">  <hr/>  <p>Page footer  </p>  </div>  </app- card >  </div> | <div class="panel panel-default">  <div class="panel-heading">  <ng-content select=".articleHeading"></ng-content>  </div>  <div class="panel-body">  <ng-content select=".articleContent"></ng-content>  </div>  </div> | |
|  | * <app-date> component is the child component in the projected component |
| **CARD COMPONENT TS** | |
| export class Template1Component implements OnInit, AfterContentInit {  @ContentChild(DateComponent, { static: true }) dateComponent: DateComponent;  constructor() { }  ngOnInit() { }  **ngAfterContentInit**(): void {  setInterval(() => {  this.dateComponent.today = new Date();  }, 1000)  } } | |

### CONTENT PROJECTION - @ContentChildren

* If we have multiple child component of projected content we can hold of them using @ContentChildren decorator

|  |  |
| --- | --- |
| **APP COMPONENT** | **CARD COMPONENT** |
| <div class="container">  <app-card>  <div class="row">  <h1>Page Heading</h1>  <hr/>  </div>  <h2 class="articleHeading">Article heading</h2>  <p class="articleContent">  **<app-date></app-date>**  **<app-date></app-date>**  <span>Article Content</span>  <div class="row">  <hr/>  <p>Page footer  </p>  </div>  </app- card >  </div> | <div class="panel panel-default">  <div class="panel-heading">  <ng-content select=".articleHeading"></ng-content>  </div>  <div class="panel-body">  <ng-content select=".articleContent"></ng-content>  </div>  </div> |
| export class Template1Component implements OnInit, AfterContentInit {  **@ContentChildren(DateComponent) dateComponents: QueryList<DateComponent>;**  constructor() { }  ngOnInit() { }  ngAfterContentInit(): void {  this.dateComponents.forEach(element => {  setInterval(() => {  element.today = new Date();  }, 1000)  });  }} |
|  | |

## ANGULAR LIFECYCLE HOOKS

When angular encounters a selector – it instatitate the component. In this creation process it go through various phases. Angular gives and opportunity to hook methods in these phases.



|  |  |
| --- | --- |
| **ngOnChange()** | This is the first life cycle hook. This gets called multiple times as well when ever any property bounded by @Input is changed |
| **ngOnInit()** | Called when the component is intitailzed. In this hook the properties are ready to be accessible. |
| **ngDoCheck()** | Called multiple times whenever it detects a change in the component’s template. e. g property value.This is called when a event is occurred , like click event |
| **ngAfterContentInit()** | Called when the projected content (ng-cotent) is initialized |
| **ngAfterContentChecked()** | Called when a change is detected in the projected content (ng-content) |
| **ngAfterViewInit()** | Called when component is fully intialialied and view is rendered |
| **ngAfterViewChecked()** | Called when component is fully intialialied and all its childs view is initialized ` |
| **ngOnDestroy()** | When we move from one page to another(Route change). The component of the previous pages is destroyed. This life cycle hook is called right before the component is destroyed. It can be used for some clean up activity |

## CREATING A SERVICE

|  |  |  |
| --- | --- | --- |
| CREATING A SERVICE(CLI COMMAND)  [SEVICES CREATED IN APP MODULE] | ng generate service <service\_name>  E.g. 🡪 ng generate service test OR ng g s <***service\_name***> | |
| * This will generate only 2 files because service are something, which contains the backend logic and has no view – so no corresponding **html or module** files are generated of it * The Services are annotated with **@Injectable** * Since the services contains the business logic which will be used other components in the module. For that we need to declare the service in component’s provider’s section * If the service is a global service – it can be added as providers app module. * ***The Service is a centralized location – where our business logic will reside – and can be shared among multiple components.*** | | |
| **GENERATED FILES**    **SERVICE**  import { Injectable } from '@angular/core';  @Injectable({  providedIn: 'root'  })  export class LoggingService {  constructor() { }  logToConsole(message:string){  console.log(message);  }  } | | **AT COMPONENT LEVEL**  import { Component, OnInit } from "@angular/core";  import { LoggingService } from "../logging.service";  @Component({  selector: "app-server",  templateUrl: "./server.component.html",  styleUrls: ["./server.component.css"],  providers: [LoggingService]  })  export class ServerComponent implements OnInit {  constructor(private loggingService: LoggingService) {}  } |
| **SERVICE INJECTED IN APP COMPONENT IN APP.MODULE.TS**  @NgModule({  declarations: [AppComponent, HelloWorldComponent],  imports: [BrowserModule, ViewModule],  providers: [TestService],  bootstrap: [AppComponent]  }) |
| **Note – Here the “**LoggingService**”** has been declared at component level and “TestService” is declared at @NgModule level . Refer *HIERARCHICAL DEPENDENCY INJECTION* for more details on it. | | |

### USING SERVICES

* The services can be used in the component using **DEPENDECY INJECTION**

|  |  |
| --- | --- |
| **SERVICE**  @Injectable({  providedIn: "root"  })  export class TestService {  constructor() {}  printToConsole(args: string) {  return args;  }  } | Let’s say we have a method in the service “printToConsole” , which we want to use /call in one of the component.  ***Angular will inject the service instance into the constructor argument as shown below*** |
| export class AppComponent {  title = "training";  constructor(private svc: TestService) {  svc.printToConsole("Hello !!");  }  } |
| Note – ***This is the new syntax for Angular 6+ version. This is same as declaring at Globally at @NgModule level . The "new syntax" does offer one advantage though: Services can be loaded lazily by Angular (behind the scenes). This can lead to a better performance and loading speed.*** | |

### HIERARCHICAL DEPENDENCY INJECTION

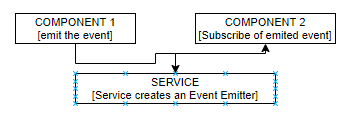
|  |  |
| --- | --- |
|  | 1. The Service can be declared as providers at each component level or at the topmost level NgModule level 2. When the service is declared at component level then component and its child component gets – the same instance of service 3. When the Service is declared as NgModule(topmost level)- Same instance of service is being used by the complete application. 4. In the diagram- App module and all its child components gets the an instance of “GlobalService” service. whereas ParentComponent and ChildComponent gets the same instance of **ParentService** service. |

### INJECTING SERVICE INTO ANOTHER SERVICE

* LoggingService has been injected in ServerService – which is being used by app server component
* **We can inject service-1 to a service-2 only if service-2 is marked as Injectable**

|  |  |  |
| --- | --- | --- |
| **SERVICE 1 [LOGGING SERVICE]** | **SERVICE 2 [SERVER SERVICE]** | **USING SERVICE** |
| import { Injectable } from "@angular/core";  @Injectable({  providedIn: "root"  })  export class LoggingService {  constructor() {}  logToConsole(message: string) {  console.log(message);  }  } | import { Injectable } from "@angular/core";  import { LoggingService } from "./logging.service";  @Injectable({  providedIn: "root"  })  export class ServerService {  constructor(private loggingService: LoggingService) {}  serverLogging(message: string) {  this.loggingService.logToConsole(message);  }  } | import { Component, OnInit } from "@angular/core";  import { LoggingService } from "../logging.service";  import { ServerService } from "../server.service";  @Component({  selector: "app-server",  templateUrl: "./server.component.html",  styleUrls: ["./server.component.css"],  providers: [ServerService]  })  export class ServerComponent implements OnInit {  constructor(private serverService: ServerService) {}  ngOnInit() {}  onCreateServer() {  this.serverService.serverLogging("Server Added=" + this.server);  }  } |

### CROSS COMPONENT COMMUNICATION USING SERVICES



* Service just creates the instance of event emitter
* This instance is levarged to emit and subscribe for the event

|  |  |  |
| --- | --- | --- |
| **SERVICE** | **COMPONENT 1 (Emits the data)** | |
| import { Injectable, EventEmitter } from '@angular/core';  import { Recipe } from './recipe.model';  @Injectable({  providedIn: 'root'  })  export class RecipeService {  private recipes: Recipe[] = [  new Recipe('Test Recipe', 'Test Description', '),  new Recipe('Test Recipe1', 'Test Description2',’’), ];  selectedRecipe = new EventEmitter<Recipe>();  constructor() { }  getRecepies() {  return this.recipes.slice();  }  } | import { Component, OnInit, Output, EventEmitter } from '@angular/core';  import { Recipe } from '../recipe.model';  import { RecipeService } from '../recipe.service';  @Component({  selector: 'app-recipe-list',  templateUrl: './recipe-list.component.html',  styleUrls: ['./recipe-list.component.css']  })  export class RecipeListComponent implements OnInit {  recipes: Recipe[] = [];  constructor(private recipeService: RecipeService) { }    showRecipeDetails(recipe: Recipe) {  this.recipeService.selectedRecipe.emit(recipe);  }  } | |
| **COMPONENT 2 (Subscribe for the data)** | | MODEL |
| import { Component, OnInit, Input } from '@angular/core';  import { Recipe } from '../recipe.model';  import { RecipeService } from '../recipe.service';  @Component({  selector: 'app-recipe-detail',  templateUrl: './recipe-detail.component.html',  styleUrls: ['./recipe-detail.component.css']  })  export class RecipeDetailComponent implements OnInit {  recipe: Recipe;  constructor(private recipeService: RecipeService) { }  ngOnInit(): void {  this.recipeService.selectedRecipe.subscribe((recipe: Recipe) => {  this.recipe = recipe;  });  }  } | | export class Recipe {  constructor(public name: string, public description: string, public imagePath: string) {  }  } |

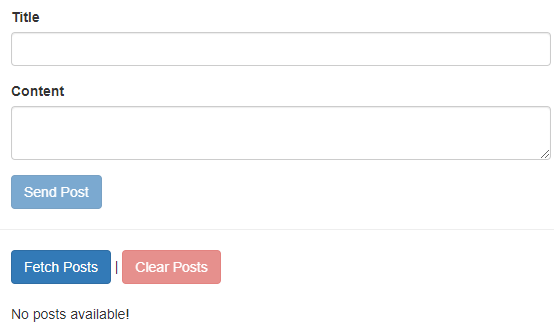
## MAKING HTTP REQUEST

### BACKEND SET UP - SETTING UP FIREBASE

|  |  |  |
| --- | --- | --- |
|  | To set up Db in firebase   1. Access the firebase url with a valid google account - <https://console.firebase.google.com/> 2. Create a firebase project using “Add Project”.Give a name to the project . 3. To Set up database in the firebase project . Click on the project name 🡪 Database 🡪 Realtime Database🡪Create Database | |
| **CREATING REALTIME DB**   1. Create database 🡪Start test mode 🡪Enable |  | |
|  | | 1. The base path of the REST API will be   [**https://angular-udemy-6a41c.firebaseio.com/**](https://angular-udemy-6a41c.firebaseio.com/)   1. The data will be stored and fetch in JSON format |

To understand the HTTP requests in Angular we will follow a common HTML template and perform different HTTP operation on the same.

SAMPLE FORM:



* This form can able to CREATE, READ , DELETE post stored in the Firebase backend DB

|  |
| --- |
| SAMPLE HTML  <div class="container">  <div class="row">  <div class="col-xs-12 col-md-6 col-md-offset-3">  <form #postForm="ngForm" (ngSubmit)="onCreatePost(postForm.value)">  <div class="form-group"> <label for="title">Title</label> <input type="text" class="form-control" id="title" required ngModel name="title" /> </div>  <div class="form-group"> <label for="content">Content</label> <textarea class="form-control" id="content" required ngModel name="content" ></textarea> </div>  <button class="btn btn-primary" type="submit" [disabled]="!postForm.valid" > Send Post </button> </form>  </div>  </div>  <hr />  <div class="row">  <div class="col-xs-12 col-md-6 col-md-offset-3"> <button class="btn btn-primary" (click)="onFetchPosts()"> Fetch Posts </button> | <button class="btn btn-danger" [disabled]="loadedPosts.length < 1" (click)="onClearPosts()" > Clear Posts </button> </div>  </div>  <div class="row">  <div class="col-xs-12 col-md-6 col-md-offset-3">  <p>No posts available!</p>  </div>  </div>  </div> |
| **SAMPLE TS FILE**  import { Component, OnInit } from '@angular/core';  import { HttpClient } from '@angular/common/http';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit {  loadedPosts = [];  constructor(private http: HttpClient) { }  ngOnInit() { }  onCreatePost(postData: { title: string; content: string }) {  // Send Http request  }  onFetchPosts() {  // Send Http request  }  onClearPosts() {  // Send Http request  }  } |

### POST REQUEST

|  |  |
| --- | --- |
| onCreatePost(postData: { title: string; content: string }) {  **this.http**  **.post(**  **'https://angular-udemy-6a41c.firebaseio.com/posts.json',**  **postData**  **) .subscribe(responseData => {**  **console.log(responseData);**  **});**  } | * post() returns an observable for which we need to subscribe * We no subsbcription is done – the request will not be sent. |

### GET REQUEST

* For any HTTP Operation we need to import HttpClientModule on the App Module level.

|  |  |
| --- | --- |
| **STEP 1 : IMPORT THE HTTP MODULE IN APP MODULE** | **STEP 2: INJECT THE SERVICES in COMPONENT** |
| import { BrowserModule } from "@angular/platform-browser";  import { NgModule } from "@angular/core";  **import { HttpClientModule } from "@angular/common/http";**  import { AppComponent } from "./app.component";  @NgModule({  declarations: [AppComponent],  imports: [BrowserModule, **HttpClientModule**],  providers: [],  bootstrap: [AppComponent]  })  export class AppModule {} | **INJECT THE SERVICES OF HttpClientModule IN COMPONENT CONSTRUCTOR**  import { Component } from "@angular/core";  import { HttpClient } from "@angular/common/http";  @Component({  selector: "app-root",  templateUrl: "./app.component.html",  styleUrls: ["./app.component.css"]  })  export class AppComponent {  title = "training";  constructor(private http: HttpClient) { }  ngOnInit() {  this.http  .get("https://api.github.com/users/avishekhsinhaRepo")  .subscribe(response => console.log(response));  }  }  **Note 🡪subscribe() is like a “then()” in promise** |

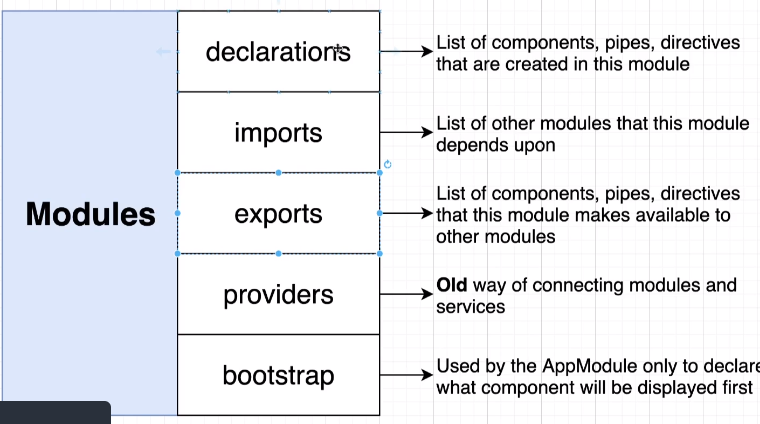
## FORMS

Angular offer2 different approaches for handling forms

1. Template Driven approach
2. Reactive approach

### TEMPLATE DRIVEN APPROACH

## ANGULAR MODULE



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1. Module can be considered as a namespace or container(like java package) 2. @NgModule 🡪 Annotation is used to define a module 3. Each Angular module has module.ts file 4. The module.ts file has a list of    1. The components the module has (declarations)    2. The Angular Services it has (providers)    3. The list of other module which it is importing (imports), we do so if we have dependency on other module. So importing module gives a way use the classes of other module | | | | @NgModule({  declarations: [  AppComponent,  HelloWorldComponent  ],  imports: [  BrowserModule  ],  providers: [],  bootstrap: [AppComponent]  }) | |
|  | | | | **CREATING A NEW MODULE (CLI COMMAND)**   * **ng generate module <module\_name>**   Ex.   * **ng generate module view**   This command will create a blank module “**view**” | |
|  | |  | **CREATING THE COMPONENT THE NEWLY CREATED MODULE**   * **ng generate component <module\_name>/<component\_name>** * **Ex -ng generate component view/address**   This CLI command will create an “address” component in the **view** module and add the entry of the “address” component in module.ts file of “view” module | | |
|  | | | Below is the hierarchy we have created so far, so if App Module want to use AddressComponent, it has to import the “view” module i.e.   1. Adding a imports entry in “**app.module.ts**” of view module 2. View Module has to expose the Address component using export | | |
| **app.module.ts (import the Module)**  @NgModule({  declarations: [  AppComponent,  HelloWorldComponent  ],  imports: [  BrowserModule,  ViewModule  ],  providers: [],  bootstrap: [AppComponent]  }) | **view.module.ts (expose the component using export)**  @NgModule({  declarations: [AddressComponent],  imports: [CommonModule],  exports: [AddressComponent]  }) | | | | **app.component.html**  <app-hello-world></app-hello-world>  <app-address></app-address> |
| Note   1. We import the module if we want to use the component of other module 2. The other module must export their components | | | | |

## ROUTING

|  |  |
| --- | --- |
| CREATING ANGULAR APP WITH ROUTING ENABLED | ng new comps –routing |
|  |  |

|  |  |
| --- | --- |
|  |  |
| **MAPPING OF URLS IN ROUTING CAN BE DONE**   * + Urls mapped to angular modules. Each mapped url will routed to a angular module   + Urls mapped to angular component. Each mapped url will routed to a angular component | |

### ROUTING TO MODULES

|  |  |
| --- | --- |
| CREATING A MODULE | ng g m <***module\_name***> --routing |

### ROUTING TO COMPONENTS

|  |  |
| --- | --- |
| Application to be developed using Route is Student Management System It will have 4 screens   1. Home Screen [StudentsComponent] 2. Add student Screen[AddStudentComponent] - User can add student name 3. List Student Screen[ListStudentComponent] – List all the added Student 4. Edit Student Screen[EditStudentComponent] – Edit a selected student |  |

### SETTING UP ROUTE

1. **CONFIGURING ROUTE IN APP MODULE**

|  |  |
| --- | --- |
| import { Routes, RouterModule } from "@angular/router";  const appRoutes: Routes = [  { path: "", component: StudentsComponent },  { path: "add", component: AddStudentComponent },  { path: "list", component: ListStudentComponent },  { path: "edit", component: EditStudentComponent }  ];  @NgModule({  declarations: [  AppComponent, StudentsComponent,  ListStudentComponent, EditStudentComponent,  AddStudentComponent  ],  imports: [BrowserModule, AppRoutingModule, RouterModule.forRoot(appRoutes)],  providers: [],  bootstrap: [AppComponent]  })  export class AppModule {}  **HTML [APP ROOT HTML]**  <div class="container">  <h2>Student Management</h2>  <ul class="nav nav-tabs">  <li routerLinkActive="active" [routerLinkActiveOptions]="{ exact: true }">  <a routerLink="/">Home</a>  </li>  <li routerLinkActive="active">  <a routerLink="add">Add Student</a>  </li>  <li routerLinkActive="active"><a routerLink="list">List Student</a></li>  <li routerLinkActive="active"><a [routerLink]="['edit'] >Edit Student</a></li>  </ul>  <div class="row">  <router-outlet></router-outlet>  </div>  </div> | 1. The **appRoutes** is an array of objects which has the configuration of Route url versus the corresponding component that will show up on this route url 2. Then finally the **appRoutes** array is added to the import section of the @NgModule of App root component   **RouterModule.forRoot(appRoutes)]**   1. The route link are further configure into HTML using **routerLink** directive 2. routerLink can also be passed as an array of string   [routerLink]="['edit']  Note –   1. When the routerLink is a relative path – it always with respect to current path – so it appends the url to the current path 2. When the routerLink is a absolute path – it always with respect to root path [e.g. - [http://localhost:4200](http://localhost:4200/)] – so it appends the url to the root path.   **ADVANTAGE OF PASSING ROUTE LINK IN AN ARRAY**   * [routerLink]="['edit'] 🡪rootPath/edit * routerLink]="['edit',’subLink’] 🡪rootPath/edit/subLink * Example:   <a class="list-group-item" \*ngFor="let user of users;let id = index" [routerLink]="['/users',id,user.name]">  URL GENERATED: **/users/2/Chris**   1. All the component will appear on the page dynamically on the page , when the route url is changed. ***router-outlet directive becomes the place holder for the component***     <router-outlet></router-outlet>   1. routerLinkActive 🡪 This takes the value of the class name which get add to the element when the give route us active. 2. routerLinkActiveOptions 🡪 |

### RELATIVE AND ABSOLUTE PATHS IN ROUTES

### NAVIGATING TO ROUTE PROGRAMATICALLY

|  |  |
| --- | --- |
| import { Router } from "@angular/router";  @Component({  selector: "app-list-student",  templateUrl: "./list-student.component.html",  styleUrls: ["./list-student.component.css"]  })  export class ListStudentComponent implements OnInit {  constructor(private studentService: StudentService, private router: Router) {}  navigateToEdit() {  // some complex logic – like DB fetch or update  this.router.navigate(["edit"]);  }  } | 1. We use navigate() function to programmatically navigate to different route 2. We follow this approach when we want to do some complex logic – before navigating. 3. The navigate() always loads the route from the root path- irrespective of – whether the path is relative or absolute |

* Since by default the navigate method always load the route with respect to root , but if we want to load the route with respect to a given route we have to pass “relativeTo” parameter to the “navigate()” method

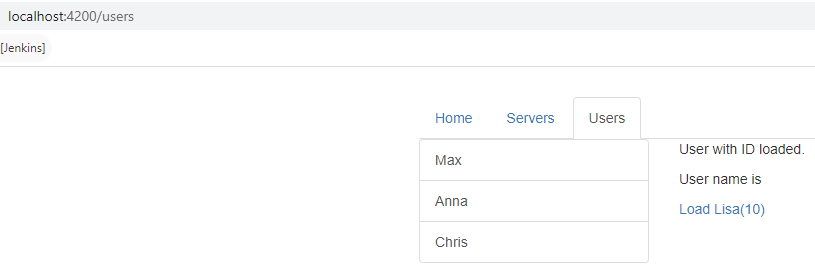
|  |  |
| --- | --- |
| **NAVIGATE METHOD - DEFAULT** | **NAVIGATE METHOD – LOADING ROUTE RELATIVILY** |
| navigateToEdit() {  this.router.navigate(["edit"]);  }  This will navigate to “edit” route with respect to root i.e  <http://localhost:4200/edit> | export class ServersComponent implements OnInit {  constructor(private serversService: ServersService, private router: Router, private route: ActivatedRoute) { }  onReloadServer() {  this.router.navigate(['servers'], { relativeTo: this.route });  }  }  This will navigate to “servers” route with respect to current route. If we are on edit route. Navigate method will take us to  <http://localhost:4200/edit>/**servers** |

### ROUTE PARAMETER PASSING IN ROUTE

|  |  |
| --- | --- |
| **CONFIGURING ROUTE in APP MODULE** | **CONSUMING PARAMETERS IN COMPONENT** |
| import { BrowserModule } from "@angular/platform-browser";  import { NgModule } from "@angular/core";  import { AppRoutingModule } from "./app-routing.module";  import { StudentDetailsComponent } from "./student-details/student-details.component";  const appRoutes: Routes = [  { path: "details/**:name**", component: StudentDetailsComponent }  ];  @NgModule({  declarations: [  AppComponent,  StudentDetailsComponent  ],  imports: [  BrowserModule,  **AppRoutingModule**,  RouterModule.forRoot(appRoutes)  ],  providers: [],  bootstrap: [AppComponent]  })  export class AppModule {} | import { Component, OnInit } from "@angular/core";  import { StudentService } from "../student.service";  import { ActivatedRoute } from "@angular/router";  @Component({  selector: "app-student-details",  templateUrl: "./student-details.component.html",  styleUrls: ["./student-details.component.css"]  })  export class StudentDetailsComponent implements OnInit {  constructor(  private studentService: StudentService,  **private route:** **ActivatedRoute**  ) {}  student = "";  studentNameParam = "";  ngOnInit() {  **this.studentNameParam = this.route.snapshot.params["name"];**  this.student = this.studentService.getStudent(this.studentNameParam);  }  } |
|  | |

### ROUTE OBSERVABLES

* The above technique to retrieve the route parameters can be when we are loading the components with the route change
* This will break when we have route parameter changing asynchronously – like loading the same route with different route parameter.



* In the above example to “Load Lisa” link will load the same route (users) with different route parameters. So it’s a kind of async operations which might happen sometime in future.
* For this kind of operation – we need a observable to handle it.

|  |  |
| --- | --- |
| import { Component, OnInit } from '@angular/core';  import { ActivatedRoute, Params } from '@angular/router';  @Component({  selector: 'app-user',  templateUrl: './user.component.html',  styleUrls: ['./user.component.css']  })  export class UserComponent implements OnInit {  user: { id: number, name: string };  constructor(private activateRoute: ActivatedRoute) { }  ngOnInit() {  this.user = {  id: this.activateRoute.snapshot.params['id'],  name: this.activateRoute.snapshot.params['name']  }  this.activateRoute.params.subscribe((user: Params) => {  this.user = {  id: user.id,  name: user.name  }  })  }  }  **HTML**  **<a [routerLink]="['/users','10','Lisa']">Load Lisa(10)</a>** | * this.activateRoute.params is a observable for which we need to subscribe to handle async change of route * Usually we need to unsubscribe from the observable in the onDestroy() life cycle hook. Since its an Angular managed observable – we don’t have to do it explicitly. Angular will do it for us. |

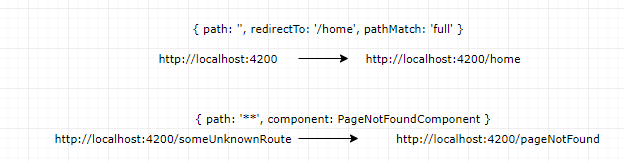
### QUERY STRING PARAMETERS AND FRAGMENTS - PASSING IN ROUTE

|  |  |
| --- | --- |
| **FROM TEMPLATE** | **OUTPUT** |
| <a class="list-group-item"  \*ngFor="let user of users;let id = index" [routerLink]="['/users',id,user.name]" **[queryParams]={admin:true,hasAccess:true}**  **fragment="loading" >**  {{ user.name }}  </a> | * The query parameters are passed as an object |
| PROGRAMATICALLY- SETTING THE QUERY PARAMETERS | export class HomeComponent implements OnInit {  constructor(private router: Router, private route: ActivatedRoute) { }  onLoadServer() {  **this.router.navigate(['/servers'],**  **{ queryParams: { allowEdit: 1 },**  **fragment: 'loading' });**  }  } |
| **RETRIVING THE QUERY STRING PARAMETERS** | |
| * The queryParams (this.route.queryParams) and this.route.fragment) are observables also – so they too can be subscribed to . | let allowEdit = this.route.snapshot.queryParams['allowEdit'];  console.log(allowEdit);  console.log(this.route.snapshot.fragment); |

### CHILD ROUTES

|  |  |
| --- | --- |
| CHILD ROUTES | SERVERS COMPONENT HTML |
| const appRoutes: Routes = [  { path: '', component: HomeComponent },  { path: 'users', component: UsersComponent },  {  path: 'servers', component: ServersComponent, children: [  { path: ':id/edit', component: EditServerComponent },  { path: ':id', component: ServerComponent },  ]  },  { path: 'users/:id/:name', component: UserComponent },  ];  @NgModule({  declarations: [  AppComponent,  HomeComponent,  UsersComponent,  ServersComponent,  UserComponent,  EditServerComponent,  ServerComponent  ],  imports: [  BrowserModule,  FormsModule,  RouterModule.forRoot(appRoutes)  ],  providers: [ServersService],  bootstrap: [AppComponent]  })  export class AppModule { } | <router-outlet></router-outlet>   1. EditServerComponet and ServerComponent is getting treated as child route of Servers Component 2. So to load the child Route – We need to add the <router-outlet> in the Parent Component Template(ServersComponent) |

### REDIRECTION AND WILD CHARACTERS IN ROUTE CONFIGURATION



|  |  |
| --- | --- |
| const appRoutes: Routes = [  { path: '', redirectTo: '/home', pathMatch: 'full' },  { path: 'home', component: HomeComponent },  {  path: 'users', component: UsersComponent, children: [  { path: ':id/:name', component: UserComponent }  ]  },  {  path: 'servers', component: ServersComponent, children: [  { path: ':id/edit', component: EditServerComponent },  { path: ':id', component: ServerComponent },  ]  },  { path: '\*\*', component: PageNotFoundComponent },  ];  @NgModule({  declarations: [  AppComponent,  HomeComponent,  UsersComponent,  ServersComponent,  UserComponent,  EditServerComponent,  ServerComponent,  PageNotFoundComponent  ],  imports: [  BrowserModule,  FormsModule,  RouterModule.forRoot(appRoutes)  ],  providers: [ServersService],  bootstrap: [AppComponent]  })  export class AppModule { } | * **The “\*\*” route must be the last route in the route configuration list** * By default, Angular matches paths by prefix. That means, that the following route will match both /recipes  and just /   { path: '', redirectTo: '/somewhere-else' }  Angular will give an error here, because that's a common gotcha: This route will now **ALWAYS** redirect you! Why?  Since the default matching strategy is "prefix" , Angular checks if the path you entered in the URL does **start with the path** specified in the route. Of course every path starts with ''  (Important: That's no whitespace, it's simply "nothing").  To fix this behavior, you need to change the matching strategy to "full" :  **{ path: '', redirectTo: '/somewhere-else', pathMatch: 'full' }**  Now, you only get redirected, if the full path is ''  (so only if you got NO other content in your path in this example). |

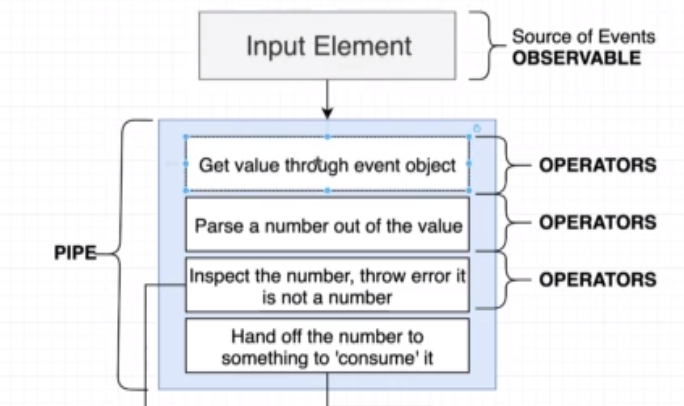
### ROUTE GUARDS

## RXJS

* It’s a separate library from Angular
* We use this instead of promise or async/await for handling async stuff.

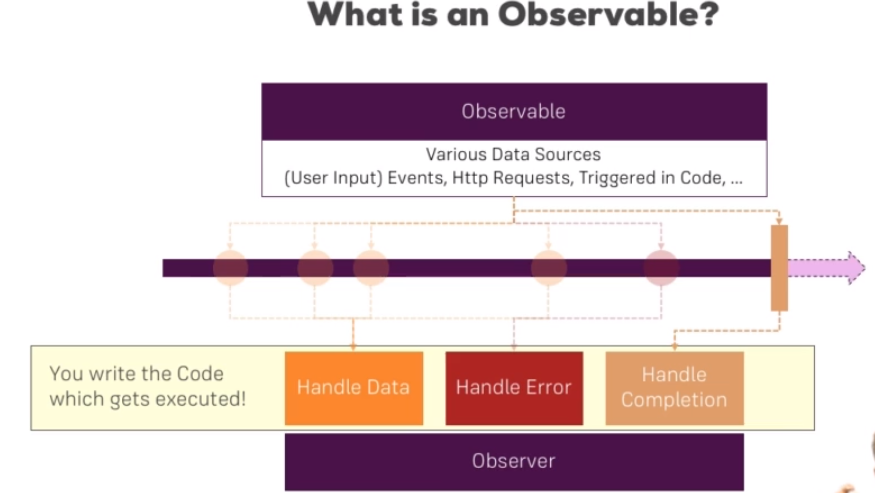
### TERMINOLOGIES

1. **OBSERVABLE**: The entity which emit the event/data
2. **OPERATOR**: The invidualal operation which act on emited data for processing
3. **PIPE**: The series of preprocessting activity is called pipes



1. **OBSERVER**:
   1. The data / event handler which finally gets the data after pre-processing.
   2. The pipe terminates in case of error and further handled by Observer as a error handler.

### OBSERVABLES



* Angular uses Observables from a third-party package RxJS
* Obserables can be consided as a data source like user input (which triggers an event), HTTP requests etc..
* Observers are the piece of logic which listens to the observables.(code in subscribe function)
* Observables are basically used to handle asynchronous tasks like user event or response from an API . Earlier we use Promises to do so.

#### OBSERVABLES – EXAMPLE –(INTERVAL)

|  |  |
| --- | --- |
| import { Component, OnInit, OnDestroy } from '@angular/core';  import { interval, Subscription } from 'rxjs';  @Component({  selector: 'app-home',  templateUrl: './home.component.html',  styleUrls: ['./home.component.css']  })  export class HomeComponent implements OnInit, OnDestroy {  private intervalSubscriber: Subscription;  constructor() { }  ngOnInit() {  this.intervalSubscriber = interval(2000).subscribe((data) => {  console.log(data);  });  }  ngOnDestroy(): void {  this.intervalSubscriber.unsubscribe();  }  } |  |

#### BUILDING CUSTOM OBSERVABLES

|  |  |
| --- | --- |
| import { Component, OnDestroy, OnInit } from '@angular/core';  import { Subscription, Observable } from 'rxjs';  @Component({  selector: 'app-home',  templateUrl: './home.component.html',  styleUrls: ['./home.component.css']  })  export class HomeComponent implements OnInit, OnDestroy {  private firstObsSubscription: Subscription;  constructor() { }  ngOnInit() {  const customIntervalObservable = Observable.create(observer => {  let count = 0;  setInterval(() => {  observer.next(count);  if (count === 5) {  observer.complete();  }  if (count > 3) {  observer.error(new Error('Count is greater 3!'));  }  count++;  }, 1000);  }); | this.firstObsSubscription = customIntervalObservable.subscribe(data => {  console.log(data);  }, error => {  console.log(error);  alert(error.message);  }, () => {  console.log('Completed!');  });  }  ngOnDestroy(): void {  this.firstObsSubscription.unsubscribe();  }  } |
| * Observer.next(data) 🡪 The data in the next method is emitted to success callback of the observer * observer.error(new Error(message)) 🡪This will terminate the observer with an error and call the error callback of the observer. * observer.complete() 🡪 This will complete the observable and calles the completion callback of the observer |
| **OBSERVABLE AND OBSERVER MAPPING** | |
|  | |

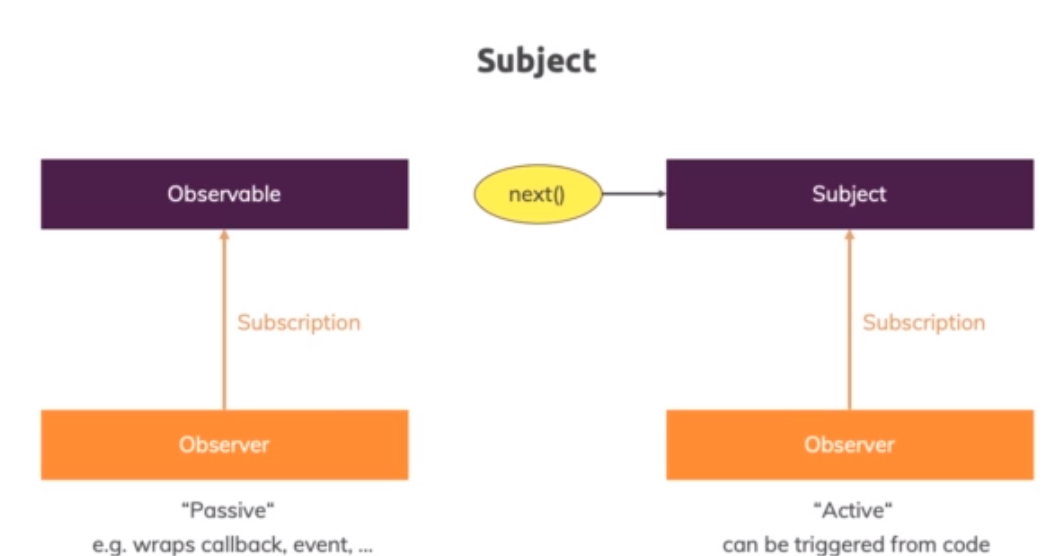
#### OPERATORS – OBSERVABLES

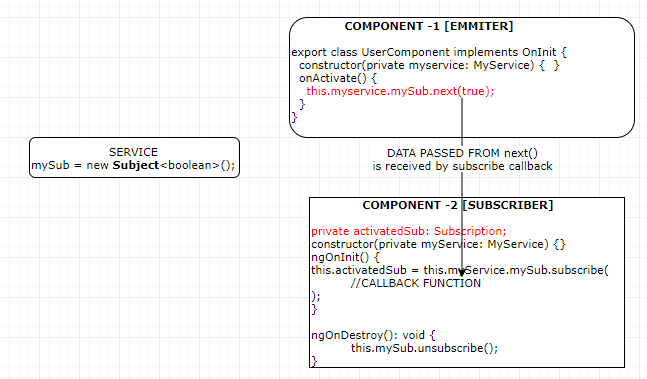


* When observable emits data. This might be possible that the subscriber needs the pre-processed data .
* The pre-processing of emited data before it is sent to obserser is done by Operators.
* List of operators : <https://www.learnrxjs.io/learn-rxjs/operators/complete>

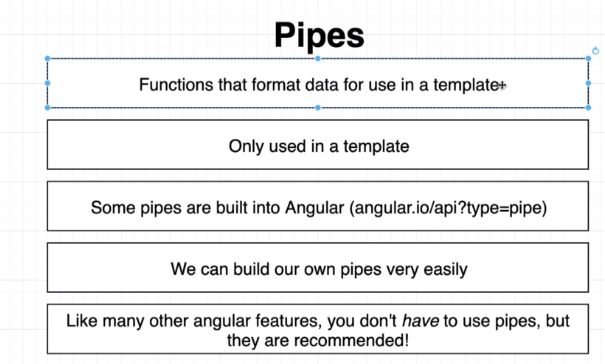
#### IMPORTANT AND COMMONLY USED OPERATORS

#### SUBJECTS - OBSERVABLES





## ANGULAR PIPES



* Angular pipes are used to transform the output in the template(HTML)
* Angular has build in pipes (as in below eample)- <https://angular.io/api?type=pipe>
* We can create custom pipes

### BUILT IN PIPES

|  |  |
| --- | --- |
| The value of username becomes an input for the “uppercase” pipe |  |
| **TS FILE**  import { Component, OnInit } from "@angular/core";  @Component({  selector: "app-simplepipe",  templateUrl: "./simplepipe.component.html",  styleUrls: ["./simplepipe.component.css"]  })  export class SimplepipeComponent implements OnInit {  constructor() {}  **todaysDate = new Date();**  ngOnInit() {}  } | **HTML**  <div class="container">  <p>{{ todaysDate | date }}</p>  <label for="userName">User Name</label>  <input type="text" [(ngModel)]="userName" name="userName" />  <p>Filtered Name: {{ userName | uppercase }}</p>  </div> |

### PARAMETERIZING AND CHAINING PIPES

* The pipes can be parameterized too using “**colon**” sign . Let’s consider an example where we want to filter the output of the Date
* todayDate = new Date() 🡪 **Sun May 17 2020 12:47:43 GMT+0530 (India Standard Time)**

|  |  |
| --- | --- |
| **PARAMETERING PIPES** | |
| **PIPE** | **OUTPUT** |
| {{ todayDate | date}} | **May 17, 2020** |
| {{ todayDate | **date:’fullDate’**}} | **Sunday, May 17, 2020** |
| **PIPE CHAINING**  In pipe chaining output of one pipe becomes input for another chained pipe. Its gets executed from left to right | |
| {{ todayDate | **date:’fullDate’|uppercase**}} | **SUNDAY, MAY 17, 2020** |

### CREATING CUSTOM PIPES

* We can create custom pipes to for some custom functionality. Let’s say we want to create a pipe which will add “…” if the number of character is more than 10 character.

|  |  |  |
| --- | --- | --- |
| **CREATING PIPES** | * **ng g p shortenpipe** * **ng generate pipe shortenpipe** | |
| **DECLARE THE PIPE IN APP MODULE** | **shortenpipe.pipe.ts** | |
| import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import {FormsModule} from '@angular/forms'  import { AppComponent } from './app.component';  **import { ShortenpipePipe } from './shortenpipe.pipe'**;  @NgModule({  declarations: [  AppComponent,  **ShortenpipePipe**  ],  imports: [  BrowserModule,  FormsModule  ],  providers: [],  bootstrap: [AppComponent]  })  export class AppModule { } | import { Pipe, PipeTransform } from "@angular/core";  @Pipe({  **name: "shortenpipe"**  })  export class ShortenpipePipe implements PipeTransform {  transform(**value**: any, ...args: any[]): any {  if (value != undefined && value.length > 10) {  return value.substring(0, 10).concat("...");  }  return value;  }  } | |
| **HTML**  <div class="container">  <p>{{ todaysDate | date: "fullDate" | uppercase }}</p>  <label for="userName">User Name</label>  <input type="text" [(ngModel)]="userName" name="userName" />  <p>  <strong>  Name: **{{ userName | uppercase | shortenpipe }}**  </strong>  </p>  </div> | |
| 1. Pipe can be declared using @Pipe decorator, which has a name property to assign a name to the property. 2. The output of **userName | uppercase becomes input for ”shortenpipe” pipe** | | |
| **PARAMETERIZING CUSTOM PIPE** | | |
| **TS FILE**  import { Pipe, PipeTransform } from "@angular/core";  @Pipe({  name: "shortenpipe"  })  export class ShortenpipePipe implements PipeTransform {  transform(value: any, limit): any {  if (value != undefined && value.length > limit) {  return value.substring(0, 10).concat("...");  }  return value;  }  } | | **HTML**  <div class="container">  <p>{{ todaysDate | date: "fullDate" | uppercase }}</p>  <label for="userName">User Name</label>  <input type="text" [(ngModel)]="userName" name="userName" />  <p> <strong>  Name: {{ userName | uppercase | shortenpipe: 10 }}  </strong> </p></div> |

**EXAMPLE 2: FILTERING OUTPUT: Filtering the output (names ) based on filter entered**

|  |  |  |
| --- | --- | --- |
| **TS FILE (PIPE)** | **COMPONENT HTML** | |
| import { Pipe, PipeTransform } from "@angular/core";  @Pipe({  name: "filternames"  })  export class FilternamesPipe implements PipeTransform {  transform(value: any, filterString: string): any {  if (filterString == undefined) {  return value;  }  const filteredNames = [];  value.forEach(element => {  if (element.indexOf(filterString) > -1) {  filteredNames.push(element);  }  });  return filteredNames;  }  } | <div class="container">  <label for="userName">Filter Name</label>  <input type="text" [(ngModel)]="**filterString**" name="userName" />  **<p \*ngFor="let uname of userNames | filternames: filterString">**  {{ uname }}  </p>  </div> | |
| COMPOENNT TS  import { Component, OnInit } from "@angular/core";  @Component({  selector: "app-simplepipe",  templateUrl: "./simplepipe.component.html",  styleUrls: ["./simplepipe.component.css"]  })  export class SimplepipeComponent implements OnInit {  constructor() {}  userNames = [];  ngOnInit() {  this.userNames.push("John");  this.userNames.push("Mike");  this.userNames.push("Baker");  this.userNames.push("Stephen");  this.userNames.push("Rob");  this.userNames.push("Tina");  }  } | **INITIAL VIEW** | **FILTERED VIEW** |

## ANGULAR MODULES & OPTIMIZATION

## ANGULAR DEPLOYMENT

## ANGULAR UNIVERSAL

### WHAT IS ANGULAR UNIVERSAL?

* **Angular universal is node-based rendering engine for Angular which is used for Server side rendering or build time pre-rendering of an Angular Application.**
* A normal Angular application executes in the browser, rendering pages in the DOM in response to user actions. Angular Universal executes on the server, generating static application pages that later get bootstrapped on the client. This means that the application generally renders more quickly, giving users a chance to view the application layout before it becomes fully interactive.

### WHY ANGULAR UNIVERSAL?

#### PERFORMANCE BENEFITS

Server rendering generates the full HTML for a page on the server in response to navigation. This avoids additional round-trips for data fetching and templating on the client since it’s handled before the browser gets a response.



This will quickly load the initial page, which will give the user a chance to quickly view the initial layout until a complete angular application becomes fully interactive.

#### SOCIAL MEDIA CRAWLERS

#### SEO (SEARCH ENGINE OPTIMIZATION)

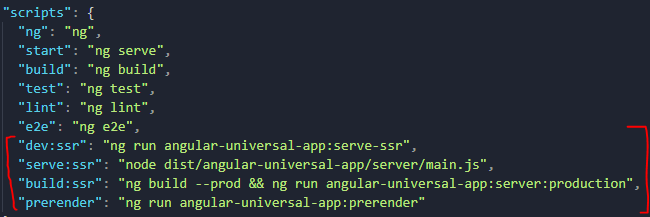
### SETTING UP ANGULAR UNIVERSAL?

|  |  |
| --- | --- |
| **CLI COMMAND TO SET UP ANGULAR UNIVERSAL** | ng add @nguniversal/express-engine |



#### IMPACT ON APP AFTER ANGULAR UNIVERSAL SET UP

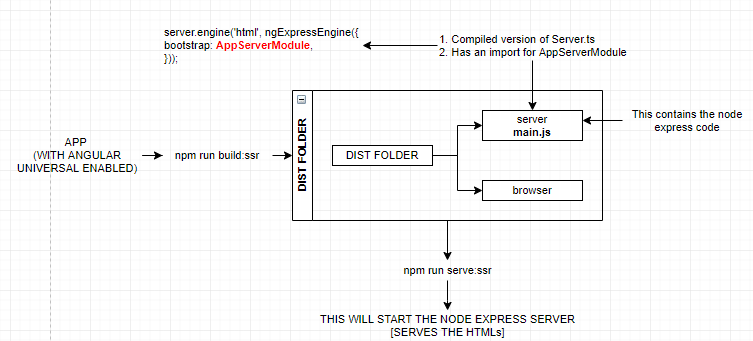
##### CHANGES IN PACKAGE.JSON



* After enabling angular universal – it adds some angular universal npm script in package.json.
* ssr: Stand for Server Side Rendering

|  |  |
| --- | --- |
| dev:ssr | * It runs the application in hot reload mode. * Used basically during development as it tracks the changes in files. * This runs the server side rendered version of the application. |
| serve:ssr | * This script start the express server – which will serve the HTML on demand |
| build:ssr | * This npm script builds the PROD version of server-side rendering on the Application. It first builds the browser version which then helps in creating the server version of the application * The output of the commands is – It creates 2 folders in the dist directory i.e “server” and “browser”. * The “browser” is client-side rendering version of the application and “server” contains the server side rendering version of the app. * The client version in-turn helps in generating the server-side version of the app. * **The main.js in in the server folder will start the Node Express server – which will be responsible for serving the HTMLs** |
| prerender |  |

##### NPM SCRIPT CHAIN – PROD BUILD



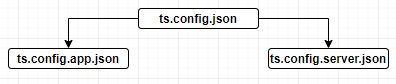
#### CHANGES IN FOLDER STRUCTURE



###### APP.SERVER.MODULE.TS

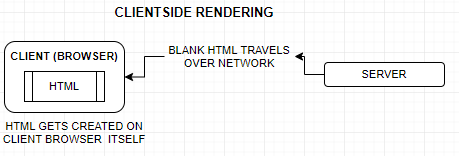
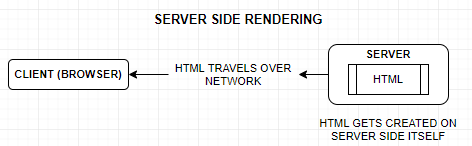
|  |  |
| --- | --- |
| **app.server.module.ts**  import { NgModule } from '@angular/core';  import { ServerModule } from '@angular/platform-server';  import { AppModule } from './app.module';  import { AppComponent } from './app.component';  @NgModule({  imports: [  AppModule,  ServerModule,  ],  bootstrap: [AppComponent],  })  export class **AppServerModule** {} | * In a general angular app . it has just one App module(app.module.ts). Enabling angular universal generates another module (app.server.module.ts) * The **AppServerModule** module imports the AppModule (this imports the entire app module component as well) and ServerModule * This actually creates two application in a angular app. A server side app and client side app. |

###### TS.CONFIG



* It creates 2 typescript config files – both inherits a base typescript config file “ts.config.json”
  + ts.config.app.json – For client-side rendering application
  + ts.config.server.json – For server-side rendering application

### SERVER AND CLIENT SIDE RENDERING AND RUNTIME INTERACTION

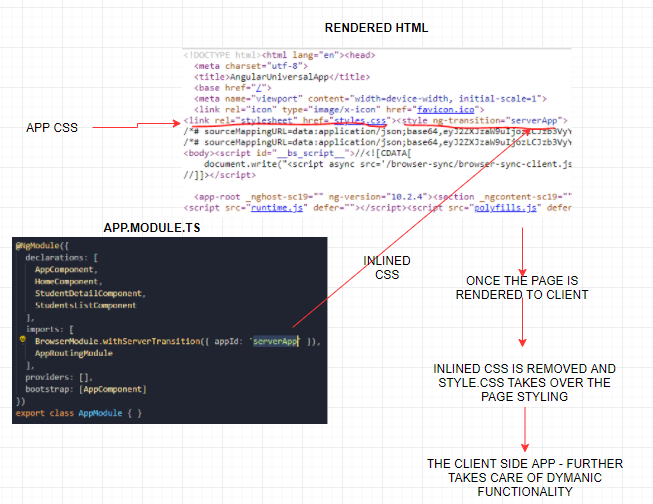


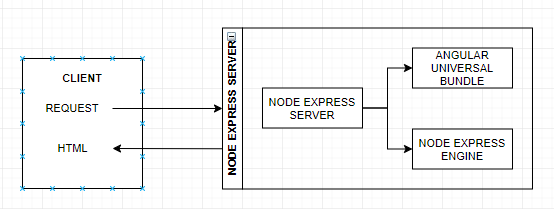
|  |  |
| --- | --- |
| **HTML RESPONSE (CLIENT-SIDE RENDERING)** | **HTML RESPONSE (SERVER-SIDE RENDERING)** |
| * In the client side rendering the HTML passed from the server is an empty HTML with <app-root></app-root> and Java script * The JS futher generates the HTML dynamically on client browser. | * In the server side rendering the entire HTML passed from the server to the client browser. It happens in the similar ways as some server side scripting language works(JSP) * No further HTML generation is needed on the client side. |

#### RUNTIME INTERACTION BETWEEN CLIENT AND SERVER-SIDE APP

1. The server-side module imports all the App module components – so the server rendering engine knows what HTML it must generates.
2. Due to the inlined css the end user never feels that the interface of app is broken.
3. The “appId” identifier helps angular to remove the inlined style from finally rendered page.

BrowserModule.withServerTransition({ **appId**: **'serverApp'** })





* Whenever a request goes from a browser – it goes to the node express server
* The node express server then make us of Angular Universal Bundle and Node Express engines to generate the HTML , which finally rendered back to the client.

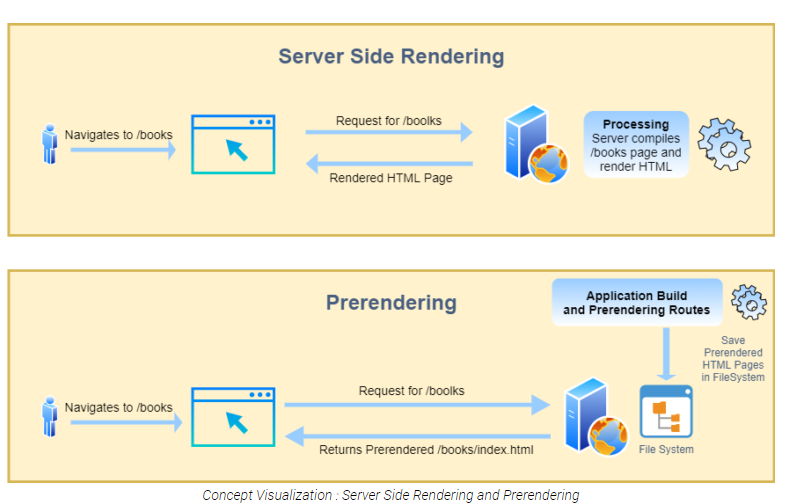
### ANGULAR UNIVERSAL – PRE- RENDERING

* Apart from getting the HTML from the server on demand. Angular universal can also be used to pre-render the HTML as well.
* **Server-Side Rendering (SSR):**  rendering a client-side or universal app to HTML on the server.
* **Prerendering**: running a client-side application at build time to capture its initial state as static HTML.

In both rendering techniques, we generate static HTML pages which can be easily crawled by crawlers.

Just the major difference is the location where HTML pages are generated. In Server-Side Rendering Technique when the user navigates to the URL => for example - /books route, the server compiles the application(render on a server) and sends the generated HTML page back to the client.

While In Prerendering Technique, All routes of the application are compiled at build time and saved as static HTML pages on file-system. These HTML pages can be served to the client using CDN. So whenever the user navigates to any route, he/she will get prerendered HTML pages.

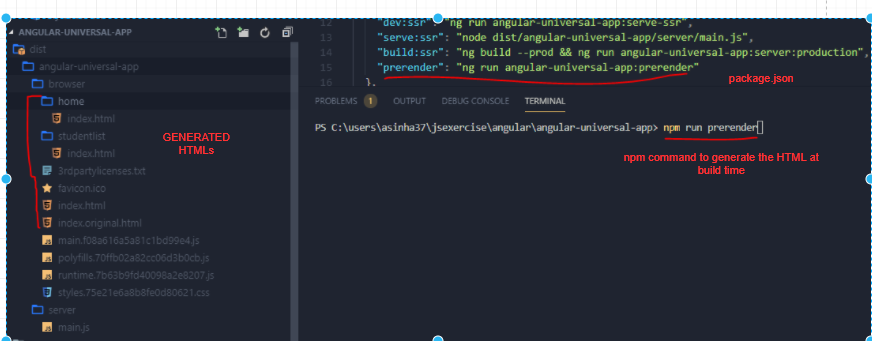


#### RUNNING PRE-RENDERING – BUILD TOOL

**NPM SCRIPT : npm run prerender**

#### CONCEPT OF PRE-RENDERING

* Usually the server-side render HTML are sent to the client on demand by the express server during the runtime.
* Pre-rendering help us to generates the HTML during the build time itself (in the dist folder). So, when we are prerendering the HTML we truly don’t need the express server. A simple HTTP server is enough to render the HTMLs
* To prerender the HTML – we need to know the all the entry points upfront of the application



* The Prerender build tool, generates the HTML along with the inline CSS as well.
* It generated the HTMLs (in the dist folder ) of static entry points. The dynamic entry points are handles differently.

#### SERVING THE PRE-RENDERED HTML TO CLIENT

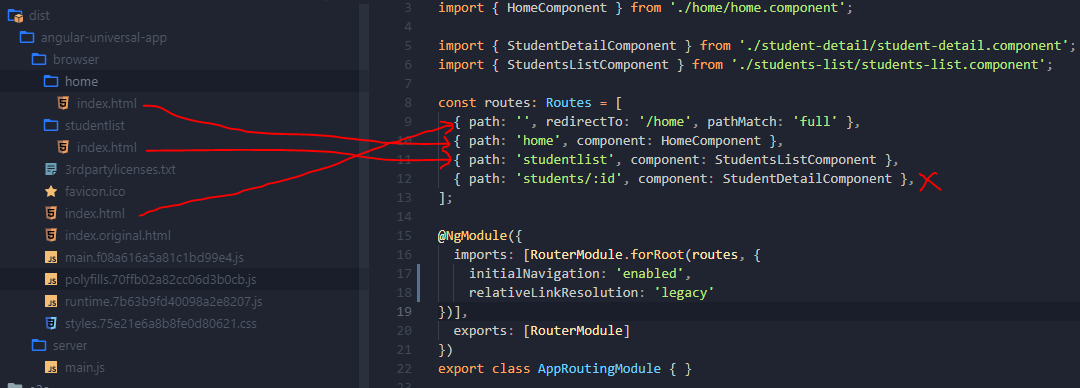
* The prerendered HTMLs can be servered by static http servers as well , rather than using a express server
* To run the HTTP server
  + Install the http-server dependency : npm i http-server
  + To run the application
    - Add the script in package.json :

**serve:prerender": "http-server -c-1 dist/angular-universal-course/browser"**

* Running the script: **npm run serve:prerender**

#### PRERENDERING ROUTES

* For routes the prerender build script generates the HTML of static routes.
* The prerender build script first scans the App routing module and first generates the HTMLs of static route
* The dynamic route prerendering is handled diffrently



#### PRERENDERING DYNAMIC ROUTES

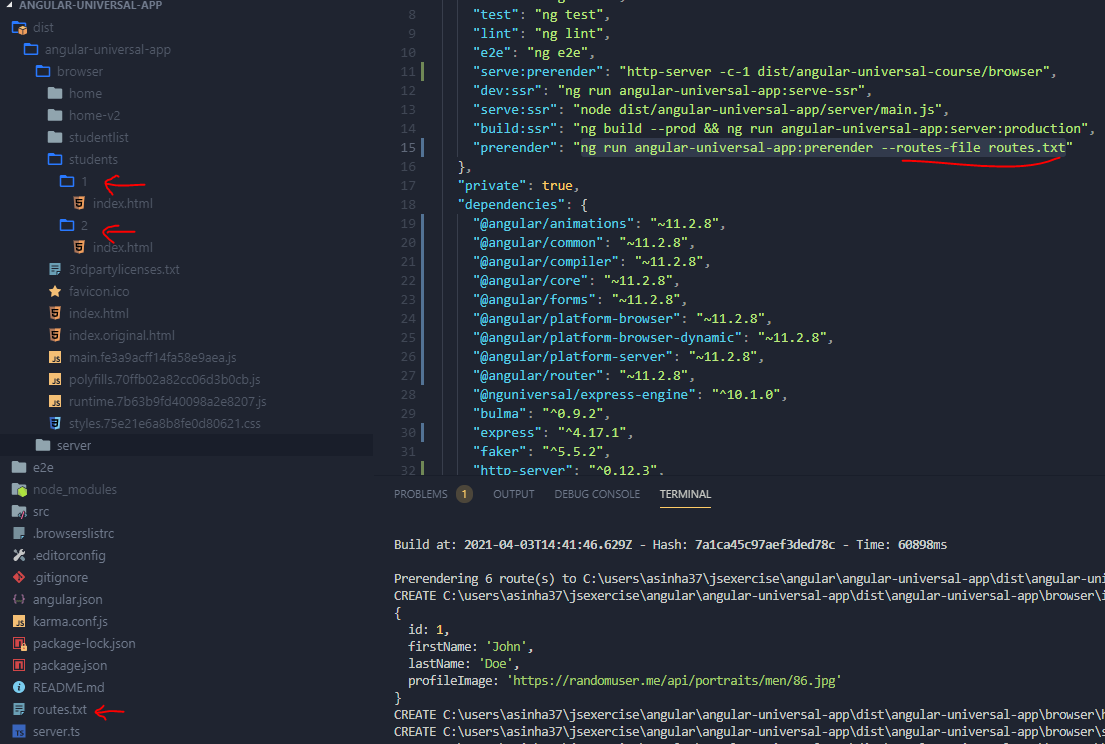
* Its easy for angular to create the prerendered HTMLs for static route. But has no way to know the HTML of the dynamic routes.
* Angular has a different way to handle dynamic routes. We must tell angular possible dynamic route upfront. The different ways to declare the dynamic routes are
  + The dynamic route can eb declared in package.json file
  + Or, dynamic routes can be configured in a text file.

##### PACKAGE.JSON CONFIGURATION

The routes can be added in pre-render npm script in package.json

##### TEXT FILE CONFIGURATION CONFIGURATION

|  |  |
| --- | --- |
| We need to create a .txt file e.g. **route.txt** in the application root directory and declare all the possible dynamic routes. |  |
| Then to generate the pre-rendered HTML run the following npm script | ng run angular-universal-app:prerender --**routes-file routes.txt** |



### APPLICATION SHELL

wde

### STATE TRANSFER API

dsf

# REACT JS

## WHAT IS REACT?

* React is JS library which is used to create user interfaces. It has component-based architecture.
* As the name suggest – Reacts to the state change.

|  |  |  |
| --- | --- | --- |
| **INSTALLING REACT** | npm install -g create-react-app | |
| **CREATING REACT PROJECT** | create-react-app <*project\_name>*  create-react-app react-complete-app |  |
| **STARTING NPM SERVER** | cd react-complete-app  npm start |
| **CHANGING THE DEFAULT PORT** | $env:PORT=5000  npm start | |
|  | We can run the command to intstall react globally and create the react app - in one go.  “npx” only works with npm version > 5.2 | |

## REACT CONCEPTS AND CREATING REACT COMPONENTS

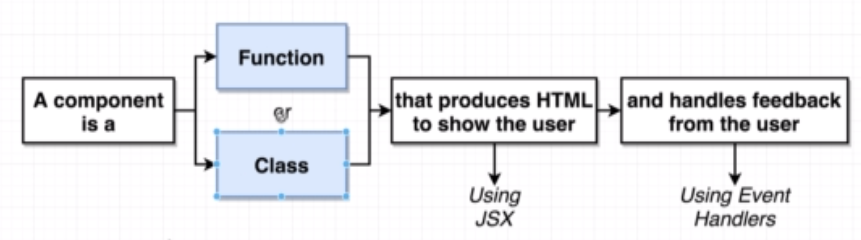
|  |  |  |  |
| --- | --- | --- | --- |
|  | | * A typical React app could be depicted as a component tree - having one root component ("App") and then a potentially infinite amount of nested child components. * When we create a react application, we build bunch of isolated, independent, and reusable components, which then composed together to build complex component. * Every React Application has once parent component called root component called “App” component. | |
|  | | * Every component in react is technically a JS class, which has   + State 🡪 The describes the state of the UI component when it will load on the page   + render() 🡪 This describes how the UI of the component will look like | |
|  | | * The react element/component is a JS object which is directly mapped with the DOM in the browser * React keep the lightweight representation of the DOM Elements in memory called Virtual DOM * When we update any react component - to keep the DOM in sync (to match the state) , it will only update that part of DOM which is mapped to that component   **THAT’S WHY IT CALLED REACT** | |
|  |  | | * The node\_modules for has all the dependencies, which we add in **package.json** file as a dependency. * In public folder has one **index.html** file which is served by the webserver. This is html file where all our react script will be injected (App root component js).   1. It has one root div which will become the parent of the react app   2. For SPA we don’t edit this html, because the root div is the place holder where we will mount our react app   3. We still can add CSS files to this file   <div id="**root**"></div> |

|  |
| --- |
| **src** - This folder has an App component which will be the parent component of all custom component we will create a in react manifest.json file give the basic functionality of PWA to react app. |

## BASIC STEPS TO CREATE REACT COMPONENT

|  |  |
| --- | --- |
| **IMPORT REACT AND REACT DOM LIBRARY** | import React from 'react';  import ReactDOM from 'react-dom'; |
| **CREATE A REACT COMPONENT (FUNCTIONAL or CLASS BASED)** | class App extends Component {  render() {  return (  <h1>Returning JSX</h1> 🡨 JSX expression  );  } }  export default App; |
| **RENDER THE COMPONENT IN THE BROWSER** | ReactDOM.render(<App /> ,document.getElementById('root')); |
|  | **OVERVIEW**   * Each component needs to return/ render some JSX code - it defines which HTML code React should render to the real DOM in the end. * JSX is NOT HTML but it looks a lot like it. Differences can be seen when looking closely though (for example className in JSX vs class in "normal HTML"). JSX is just syntactic sugar for JavaScript, allowing you to write HTMLish code instead of nested React.createElement(...) calls. * The components we create are injected in App component (as a child component) and finally the App component script is injected in index.html file. * Hence the components will be child of App component. * The class based react component can be created by extending Component * It has render method which has to return valid JSX expression (JSX looks very similar to HTML, but it’s not) * The JSX expression then gets compiled into JS (its equivalent createElement() method) |

## OVERVIEW : CREATING REACT COMPONENT



|  |  |
| --- | --- |
| 1. **FUNCTIONAL COMPONENTS** : referred to as "presentational", "dumb" or "stateless" components | const cmp = () => { return some JSX } |
| 1. **CLASS-BASED COMPONENTS**: referred to as "containers", "smart" or "state-full" components   It has render method which has to return valid JSX expression (JSX looks very similar to HTML , but it’s not).The JSX expression gets compiled into JS | class Cmp extends Component { render () {  return some JSX  } } |

## UNDERSTANDING JSX

* The JSX expression looks very much like HTML. The JSX expression is finally compiled to JS [<https://babeljs.io/repl>]

JSX COMPILATION

|  |  |  |  |
| --- | --- | --- | --- |
| **JSX** | | **COMPILED JSX** | |
| <h1 className="heading">Hello</h1> | | React.createElement("h1", { className: "heading"}, "Hello"); | |
| <p className="heading"><span className="spanStyle">Hello</span></p> | | "use strict";  React.createElement("p", { className: "heading" }, React.createElement("span", { className: "spanStyle"}, "Hello")); | |
| * The JSX expression compiled as a JS code. The compiled JSX has createElement method which created the DOM elements in the browser. So we can use React.createElement method too to render the HTML element. * ***We use “className” instead class, because the class is a reserved word in JS*** | | | |
| **UNDERSTANDING React.createElement METHOD** | | | |
| EXAMPLE : - **React.createElement('h1', {}, 'My First React Code');** | | | |
| * The first one is the type of element we're creating, in this case an <h1> tag. This could also be another React component. If we're creating an HTML element, we pass in the name as a string, just like we did above. If we're creating a React component, we pass in the variable that the component is assigned to. * The second argument is an object containing properties ('props' in React terms) that get passed to the component.. * Finally, the last argument is the children of that component. This can be a quoted string like shown above, in which case the content will be interpreted as text. However, we can also pass in a reference to another component, allowing us to nest elements and components within each other (we'll get to that in a bit). * To render the component. We do this using ReactDOM.render(). This takes two arguments: the first one being the thing we want to render (our title element), and the second one is a target DOM node to render things into. | | | |
| **APP JS** | **INDEX.JS** | | **INDEX.HTML** |
| import React, { Component } from "react";  class App extends Component {  **render**() {  **return** React.createElement("div", { className: "heading" }, "Test Data");  }  }  export default App; | import React from 'react';  import ReactDOM from 'react-dom';  import App from './App';  ReactDOM.render(<App />, document.getElementById('root')); | | <div id="root">  </div> |
|  | | |

JSX REFERING JS VARIABLES

|  |  |
| --- | --- |
| **JSX CAN ALSO REFER THE JAVASCRIPT VARIABLE** | const App = () => {  const message = "Hello World";  return <div><h1 >{message}</h1> </div>;  }; |
| **JSX CAN ALSO REFER THE JAVASCRIPT FUNCTIONS** | function getMessage() {  return "Hello World!";  }  const App = () => {  return <div><h1>{getMessage()}</h1> </div>;  }; |

JSX RESTICTIONS

* We should not use Javascript reserved word. For example we use “className” instead class , because the class is a reserved word in JS.
* There should be one parent element in a JSX expression. But there is a way to overcome this limitation

|  |  |
| --- | --- |
| **VALID** | **INVALID** |
| <div>  <p>Hello</p>  <p>World</p>  </div> | <div>  <p>Hello</p>  </div>  <p>World</p> |

## CREATING A DEMO REACT APP – BLOG POST APP

|  |  |  |
| --- | --- | --- |
|  | | Note : We will be using a third party for   * **CSS**: <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/semantic-ui/2.4.1/semantic.min.css"> [TO BE ADDED IN index.html] * **AVATAR IMAGES**:   + <img src="https://source.unsplash.com/random" />   + OR BY USING FAKER NODE MODULE : <https://www.npmjs.com/package/faker>     - To install the node module : npm i faker * HTML : <https://semantic-ui.com/views/comment.html>   To create the blogpost component   1. We will one Parent Componet (Comments) . 2. The Parent componet has an **approval form component** 3. The Approval Component has child componenst “CommentDetails” . |
|  |  |
| **CONNECTING THE COMPONENTS**  **EXPORT**  export default CommentDetails;  **IMPORT**  import CommentDetails from './CommentDetails';  (This is the relative path of the JS file without having .js extension) |  |

### COMMUNICATION BETWEEN COMPONENTS

|  |  |
| --- | --- |
|  | * React uses a “props” system to communicate between a Parent component to child/ nested components. |

### CODE IMPLEMENTATION

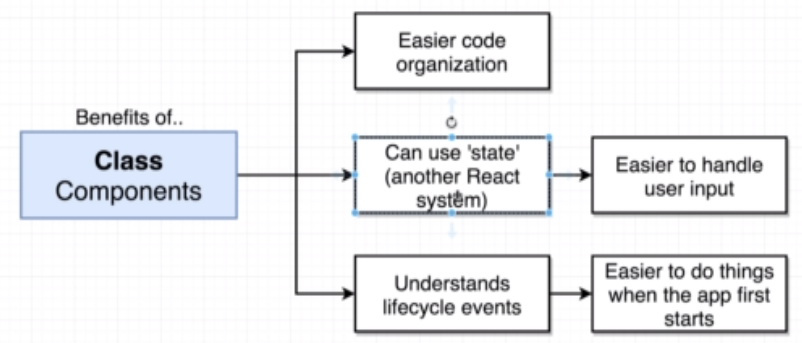
|  |  |
| --- | --- |
| **Index.js**   * The App Component is a functional component which as Child Component “ApprovalCard” component * The ApprovalCard has in turn one more child component “CommentDetails “. | import React from 'react';  import ReactDOM from 'react-dom';  import faker from 'faker';  import CommentDetails from './CommentDetail';  import ApprovalCard from './ApprovalCard';  const App = () => {  return <div className="ui container comments">  <h3 className="ui dividing header">Comments</h3>  <div class="ui cards">  <ApprovalCard>  <CommentDetails name={faker.name.firstName()} />  </ApprovalCard>  <ApprovalCard>  <CommentDetails name={faker.name.firstName()} />  </ApprovalCard>  </div>  </div>;  };  ReactDOM.render(<App />, document.getElementById("root")); |
| **ApprovalCard.js**   * The ApprovalCard component can access its child component using “{props.children}” | import React from 'react';  const ApprovalCard = (props) => {  return <div className="card">  <div className="content">  {props.children}  </div>  <div className="extra content">  <div className="ui two buttons">  <div className="ui basic green button">Approve</div>  <div className="ui basic red button">Decline</div>  </div>  </div>  </div>;  };  export default ApprovalCard; |
| **CommentDetail.js**   * The value pass as an attribute from the parent can be accessed using “{props.*attributeName*}” | import React from 'react';  const CommentDetail = props => {  return <div className="comment">  <a href="/" className="avatar">  <img alt="avatar" src="https://source.unsplash.com/random" />  </a>  <div className="content">  <a className="author">{props.name}</a>  <div className="metadata">  <span className="date">Today at 5:42PM</span>  </div>  <div className="text">  How artistic!  </div>  <div className="actions">  <a className="reply">Reply</a>  </div>  </div>  </div>;  }  export default CommentDetail; |

## CLASS BASED COMPONENTS

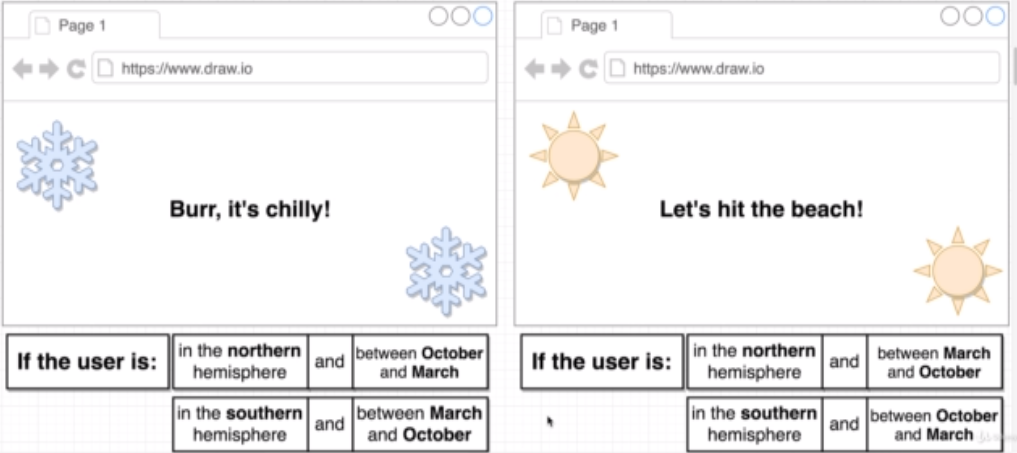
**HOW FUNCTIONAL COMPONENT AND CLASS COMPONENT ARE DIFFERENT?**

|  |  |
| --- | --- |
|  | * In early phase of react – functional componenst was only responsible for rendering the JSX. * It didn’t had access to life cycle methods and state system * Earlier - The Class based component can able to access life cycle hooks and had capability to do state management * With recent version of React - Functional components has become powerful due to ***Hook System which enabled it to manage the component’s state as well.*** |
|  |

### BENEFITS OF CLASS BASED COMPONENT



### BUILD APP USING CLASS BASED COMPONENT



***We will be building a demo app to see the usability of Class based component***

**APP OVERVIEW -** The UI will show up a season information based on the User’s Geolocation and the Month.

|  |  |
| --- | --- |
| **STEP NEED TO CREATE THE DEMO APP** | **SOLUTION** |
|  | **Step 1: USER PHYSICAL LOCATION**  To fetch the Geolocation we will be using the Geo location API (<https://developer.mozilla.org/en-US/docs/Web/API/Geolocation_API> )  **EXAMPLE**  window.navigator.geolocation.getCurrentPosition(  position => console.log(position), 🡨 SUCCESS CALLBACK  error => console.log(error.message) 🡨 ERROR CALLBACK  ); |
| In the browser we will be getting a pop-up which ask for a permission to allow the location. If permission granted – The code will end-up with success call back – other wise error callback |

**GOTCHAS IN GETTING THE USER LOCATION**

|  |  |
| --- | --- |
|  | **FUNCTIONAL COMPONENT**  import React from 'react';  import ReactDOM from 'react-dom';  const App = () => {  window.navigator.geolocation.getCurrentPosition(  position => console.log(position),  error => console.log(error.message)  );  return <h1>Latitude:</h1>;  };  ReactDOM.render(<App />, document.getElementById('root')); |
|  | **WHAT EXTRA WE GET WITH CLASS BASED COMPONENT?**  ***There is an issue with this implementation – Getting the geolocation is an asych operation. So, by the time the API gives the response, the functional component will return the JSX. There is no way for a functional component to re-render the component again with the updated Latitude value. Due to incapability of State Management the CLASS BASED COMPONENT is the right choice*** |

### CREATING CLASS BASED COMPONENTS

|  |  |
| --- | --- |
|  | **EXAMPLE (Person.js)**  import React from 'react';  import ReactDOM from 'react-dom';  class Person extends React.Component {  render() {  return (  <div>  <p>My name is {this.props.firstName} {this.props.lastName}</p>  </div>  );  }  } |

#### STATE MANAGEMENT IN CLASS BASED COMPONENT

RULES OF STATE

1. The state is a JavaScript object contains the data relevant to a component
2. State is usable only with class-based components.
   1. Note –With the recent React development state can be updated too using functional component using “hooks” system.
3. **Updating the state causes the component to re- render (render method is called as many times we change the state)**
4. **The State must be initiazed when the component is created (May be in constructor)**
5. **STATE CAN ONLY BE UPDATED USING THE FUNCTION setState()**

#### STATE MANAGEMENT IN SEASONS APP

|  |  |
| --- | --- |
| import React from 'react';  import ReactDOM from 'react-dom';  class App extends React.Component {  constructor(props) {#1  super(props);  this.state = { lat: null, errorMessage: '' };#2  window.navigator.geolocation.getCurrentPosition(  position => {  this.setState({ lat: position.coords.latitude }); #3  },  error => {  this.setState({ errorMessage: error.message }); #4  }  );  }  render() {  if (this.state.errorMessage !== '' && this.state.lat === null)  return <h1>Error: {this.state.errorMessage}</h1>; #6  else if (this.state.lat !== null && this.state.errorMessage === '')  return <h1>Latitude: {this.state.lat}</h1>; #5  return <h1>Loading!</h1>;  }  }  ReactDOM.render(<App />, document.getElementById('root')); | * #1 🡪 The class-based component has constructor which has “props” value which in turn calls the super(props) * #2 🡪 Intiallization of state object (in constructor)   #3 🡪   * The setState() function is called when response is received * It updates the state of the component and re- renders the component as well. * #4 – Set State in error scenario * #5 - Updates the value when the state of the “lat” property is updated. * In render method we are doing the condition rendering |

### LIFE CYCLE METHOD OF A CLASS BASED COMPONENT

|  |  |
| --- | --- |
|  | 1. The class based component go through a series of life cycle methods during the rendering 2. **componentDidMount**() 🡪 It gets called once when the component is visible on the screen 3. **componentDidUpdate() 🡪 It get called everytime the state of the component is updated i.e when the component’s setState() method is called.** 4. **As the render() method is called also as number of times we update the state of the component.** |
|  |

#### RESPONSIBILITIES OF LIFE CYCLE METHODS

|  |  |
| --- | --- |
|  | |
| **componentDidMount** | * This is the right spot to get the initial data loading like initiating the XHR call |
| **componentDidUpdate** | * This method is called whenever we update the state of the component * Usability can be   + making a XHR call with every user input or auto complete feature.   + Components getting a new set of **props** from the parent component |
| **Render** | This method should have responsibility to return a JSX |
| **componentWillUnMount** | To some clean up |

**CODE REFACTORING – LEVARAGING LIFE CYCLE METHODS**

|  |  |
| --- | --- |
| class App extends React.Component {  constructor(props) {  super(props);  this.state = { lat: null, errorMessage: '' };  window.navigator.geolocation.getCurrentPosition(  position => {  this.setState({ lat: position.coords.latitude });  },  error => {  this.setState({ errorMessage: error.message });  }  );  }  render() {  if (this.state.errorMessage !== '' && this.state.lat === null)  return <h1>Error: {this.state.errorMessage}</h1>;  else if (this.state.lat !== null && this.state.errorMessage === '')  return <h1>Latitude: {this.state.lat}</h1>;  return <h1>Loading!</h1>;  }  } | class App extends React.Component {  state = { lat: null, errorMessage: '' };  componentDidMount() {  window.navigator.geolocation.getCurrentPosition(  position => this.setState({ lat: position.coords.latitude }),  error => this.setState({ errorMessage: error.message })  );  }  render() {  if (this.state.errorMessage !== '' && this.state.lat === null)  return <h1>Error: {this.state.errorMessage}</h1>;  else if (this.state.lat !== null && this.state.errorMessage === '')  return <h1>Latitude: {this.state.lat}</h1>;  return <h1>Loading!</h1>;  }  } |
| * In the code refactoring the XHR call in the constructor has neen moved to componentDidMount() method * The initialization of state object can be done without constructor without using “**this**”. They are identical | |

#### DEFAULT PROPERTY

|  |  |
| --- | --- |
|  | * The userName props can be passed from the Parent component . * But If it is not passed – it will take the value passed in the “defaultProps” |

### ADDING CSS TO COMPONENT

#### EXTERNAL CSS

|  |  |  |
| --- | --- | --- |
| **FOLDER STRUCTURE** | **IMPORT IN COMPONENT** | **HTML** |
|  | import React, { useState } from "react";  import "./Person.css";  const person = () => {  const [personState, setPersonState] = useState({  firstName: "Max",  lastName: "Doe"  });  return (  <div>  <p className="personTitle">  My name is {personState.firstName} {personState.lastName}  </p>  </div>  );  };  export default person; | * When we add any css in the component level it is automatically added in head on the HTML(globally) |
| **Person.css**  **.personTitle {**  **color: red;**  **}** |

#### INLINE CSS

|  |  |
| --- | --- |
| import React, { Component } from "react";  class Dog extends Component {  state = {  color: "brown",  breed: "Labra"  };  render() {  const inlineStyle = {  color: "red",  backgroundColor: "green"  };  return (  <div>  <h1 style={inlineStyle}>  Dog is {this.state.color} in color and of {this.state.breed} Breed  </h1>  </div>  );  }  }  export default Dog; | * The inline styling can be given using a JS object and finally using the object as style={inlineStyle} , on HTML element * Note the CSS in the JS must be given in camel case e.g.   background-color attribute in CSS is written as backgroundColor in JS. |

### OPERATORS

#### CONDITIONAL OPERATOR

|  |  |
| --- | --- |
| **USING TERNARY OPERATOR**  import React, { Component } from "react";  class Dog extends Component {  state = {  color: "brown",  breed: "Labra",  showContent: true  };  toggleContent = () => {  const currentState = this.state.showContent;  this.setState({  showContent: !currentState  });  };  render() {  return (  <div>  { this.state.showContent ? (  <h1>  Dog is {this.state.color} in color and of {this.state.breed} Breed  </h1>  ) : null}  <button onClick={this.toggleContent}>Click</button>  </div>  );  }  }  export default Dog; | * We are |

### FINAL APP DESIGN AND CODE

##### VIEW

GITHUB REPO : <https://github.com/avishekhsinhaRepo/React-Applications.git>

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

## HANDLING EVENTS AND USER INPUTS

Application Name : Image Search

|  |  |
| --- | --- |
|  |  |

### BINDING EVENTS

|  |  |
| --- | --- |
| class SearchBar extends React.Component {  onInputChange(event) {  console.log(event.target.value);  }  render() {  return (<div className="ui segment"><form className="ui form" >  <div className="field">  <label>Image Search</label>  <input type="text" name="first-name" placeholder="Search" onChange={this.onInputChange} />  </div>  </form></div>);  }  }  export default SearchBar; | **Note – We do not put “()” while calling the event handler – otherwise it will be invoked when the component is loaded.** |

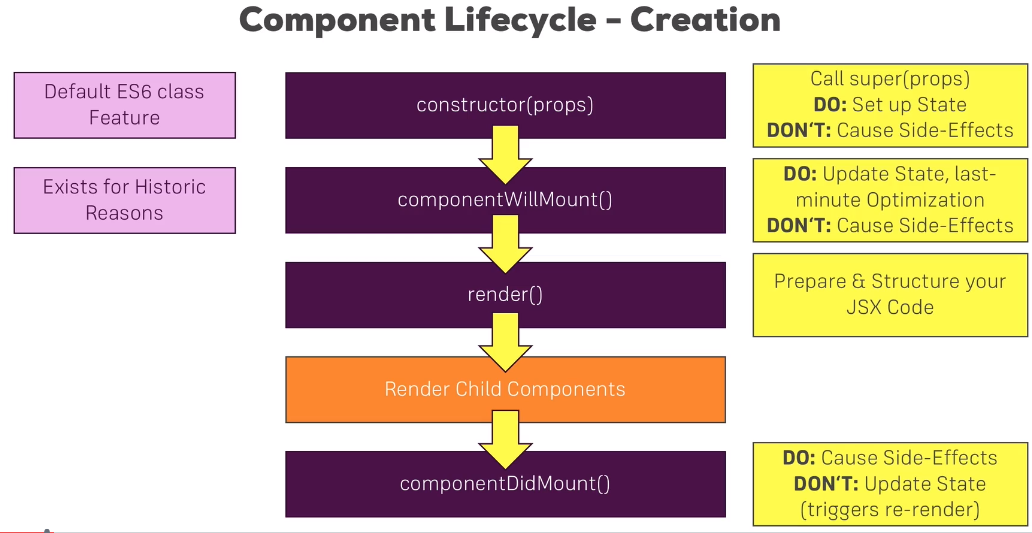
#### CONTROLLED VERSUS UNCONTROLLED INPUTS

## HTTP REQUEST IN REACT

* In React paradigm the HTTP request can be accomplished using
  + XMLHTTPRequest – Native JS Code
  + Third party library called **Axios**

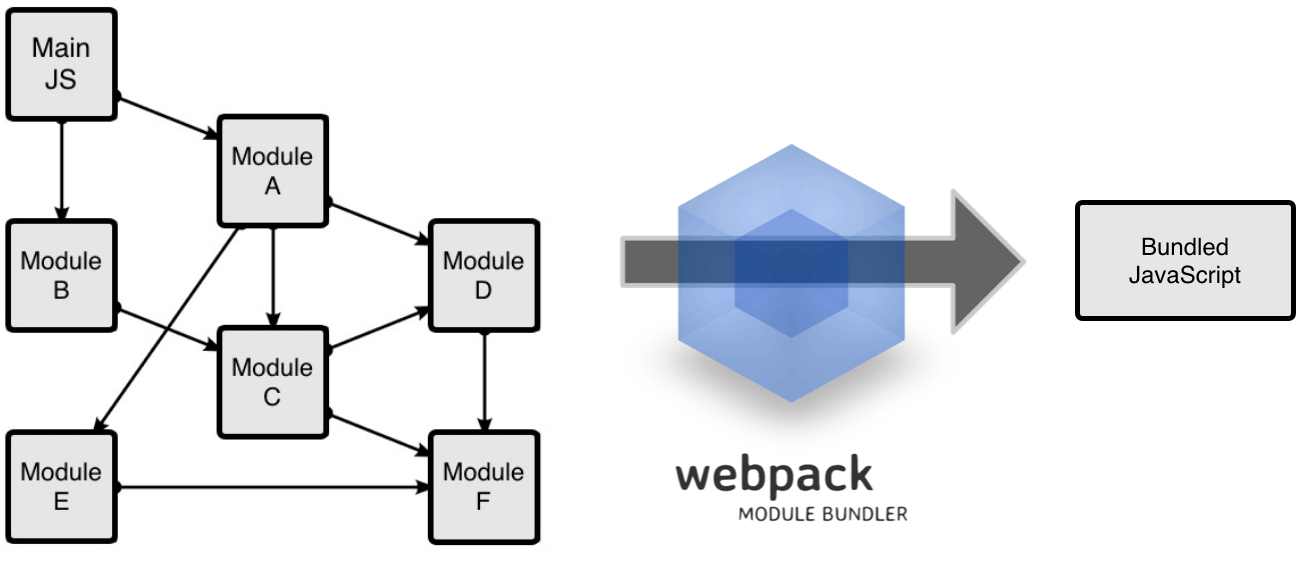
|  |  |
| --- | --- |
| INSTALLING AXIOS | npm install axios --save |

**WHEN TO IMPLEMENT HTTP REQUEST**



# WEB PACK

* WEBPACK is a module web bundler.
* The webpack becomes more important for SPA’s because- This need a huge amount of Javascript code , which is responsible for dynamically showing up the HTML/Content on client browser unlike server side rendering(HTML is rendered by the server itself)



## MODULES IN JAVASCRIPT

* A module is just a file. One script can be one module.
* For large application – we logically divide the application based on functionality. Those logical divisions are called Modules.
* ***A module is a reusable piece of code that encapsulates implementation details and exposes a public API so it can be easily loaded and used by other code.***

**WHY MODULES?**

* **ABSTRACTION** : To delegate functionality to specialized libraries so that we don't have to understand the complexity of their actual implementation
* **ENCAPSULATION: T**o hide code inside the module if we don't want the code to be changed
* **REUSABILITY**: To avoid writing the same code over and over again
* **DEPENDENCY MANAGEMENT**: To easily change dependencies without rewriting our code

**PROBLEMS WITH MODULES**

|  |  |  |
| --- | --- | --- |
|  |  | * When we break the Javascript in modules we need to load the dependent modules should be load prior .So order of loading the modules on the page is important * **WEBPACK helps us such kind of dependency management among the modules** |
| ***Along with dependency management - Webpack also bundle all the modules together into a single file hence, minimize the number of network requests.*** |

**WEBPACK IN ACTION**

To understand webpack let build a small project to leverage webpack. We will follow below steps to do so

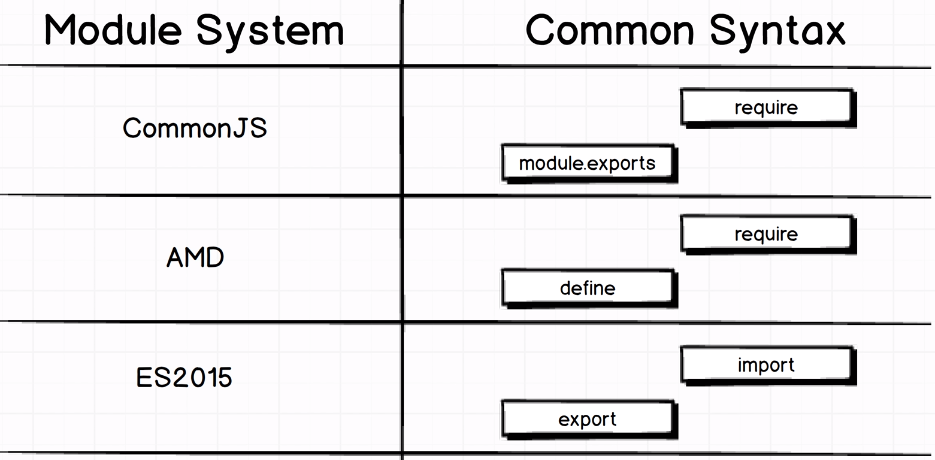
1. **Create new NPM project**
2. **Create 2 JS modules**
3. **Install and configure webpack**
4. **Run webpack**

|  |  |  |
| --- | --- | --- |
| **STEP 1: CREATING NPM PROJECT** | **npm init**  This will create a npm project with package.json file. This file is used for dependency management and configure sone build script | |
| **STEP 2: CREATING JS MODULES** |  | Here index.js is a dependent file which will call the function fron sum.js |
|  |

**After we create modules (as above) we need to link them up. So the way we can link all the modules we need to follow some pattern, which is know as MODULE FORMAT or MODULE SYSTEM**

### MODULE FORMATS/SYSTEM

* IN ES5 we didn’t have concept of modules, so developer came up with different approaches to modularize the code. The pattern/syntaxes to define a module are called Module Formats. The popular module formats used in ES5 are
  + **AMD**
    - Asynchronous Module Definition.
    - Loads the modules asynchronously
  + **COMMON JS**
    - This module system is used by Node JS
  + **ES 2015** – Universal Module definition – Used in Browsers and Node JS



### IMPLEMENTING COMMON JS MODULE FORMATS

|  |  |  |
| --- | --- | --- |
| sum.js (Module 1) | Index.js(Module 2) | |
| const sum = (a, b) => a + b;  **module.exports = sum;** | **const sum = require("./sum");**  const total = sum(10, 5);  console.log(total); | * All the export statement go at the bottom of the module * The “require” statement go at the top of the module * **The “require” function always takes the relative path of the module which it is importing** |

### INSTALLING AND CONFIGURING WEBPACK

|  |  |
| --- | --- |
| **INSTALLING WEBPACK (2.0) AS DEV DEPENDENCIES** | **npm install --save-dev webpack** |
| This will add web pack as project dependencies in package.json file | {  "name": "webpack-exercise",  "version": "1.0.0",  "description": "",  "main": "index.js",  "scripts": {  "test": "echo \"Error: no test specified\" && exit 1"  },  "author": "",  "license": "ISC",  **"devDependencies": {**  **"webpack": "^2.7.0"**  **}**  } |
| **CREATING WEBPACK CONFIG FILE**   * To know the webpack know regarding the bundling, entry point etc..configuration – we need to create a file names ”**webpack.config.js**” * This is kind of instruction file for webpack. |  |
| **webpack.config.js**  const path = require("path");  const config = {  **entry: "./src/index.js",**  output: {  **path: path.resolve(\_\_dirname, "build"),**  **filename: "bundle.js"**  }  };  module.exports = config; | 1. The entry property give the entry point JS file. We need to provide the relative path (from project directory) of the script file 2. Output property – This property will help us to configure the output file 3. **Path property** – To configure path of the o/p file . 4. This is the absolute path .We are using node to get the absolute path of the project(path.resolve()) 5. “build” is the name of the parent folder where “**bundle.js**” will be created 6. **\_\_dirname is a node variable which is the path of current working directory.** |

### INSTALLING AND CONFIGURING WEBPACK

|  |  |  |
| --- | --- | --- |
| UPDATE PACKAGE.JSON | RUNNING WEBPACK | INCLUDING THE FILE IN HTML |
| {  "name": "webpack-exercise",  "version": "1.0.0",  "description": "",  "main": "index.js",  **"scripts": {**  **"build": "webpack"**  **},**  "author": "",  "license": "ISC",  "devDependencies": {  "webpack": "^2.2.0-rc.0"  }  } | **npm run build**  This will create a bundle.js file in a bundle folder | <html class="no-js">  <head></head>  <body>  **<script src="./build/bundle.js"></script>**  </body>  </html> |
|  |

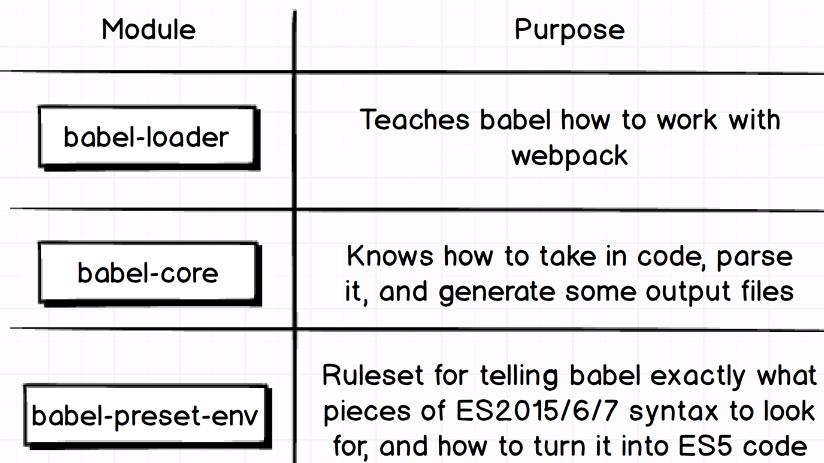
### MODULE LOADERS IN WEBPACK

* **Loader are used to do some pre-processing before bundling the code**
* Preprocessing like transpiling ES6 code(Babel Loader), preprocessing CSS and images

#### BABEL LOADER

**Babel loader transpiles the ES6 code to ES5 code**

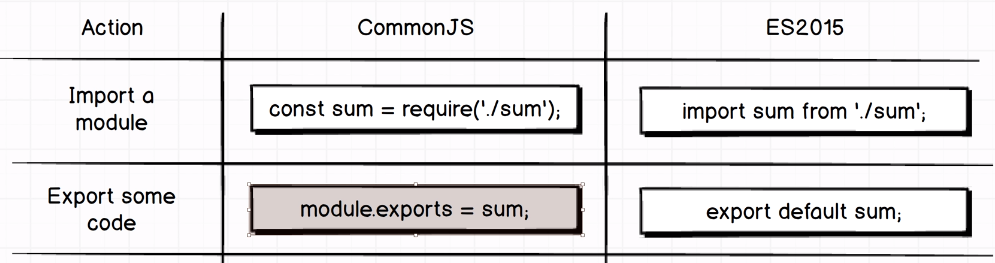
1. Babel Loader – Babel support multiple build sytem.This helps in compartibility with the build system as webpack
2. Babel Core –
3. Babel preset env – ES6 🡺 ES5



|  |  |  |  |
| --- | --- | --- | --- |
| **INSTALLING BABEL LOADER** | npm install --save-dev **babel-loader @babel/core @babel/preset-env** | | |
|  | | | |
| * To configure any loader we have to tell webpack on which type of files loader has to be applied. It can be applied to any type of files like JS, CSS, images etc. * We provide this configuration in webpack.config.js using some regex | | | |
| **CONFIGURATIONS** | | | |
| **STEP 1 :**  **CONFIGURATION IN WEBCONFIG JS**  const path = require("path");  const config = {  entry: "./src/index.js",  output: {  path: path.resolve(\_\_dirname, "build"),  filename: "bundle.js"  },  module: {  rules: [  {  **use: "babel-loader",**  test: **/\.js$/**  }  ]  }  };  module.exports = config; | | **STEP 2 :**  We need to create a file with name **.babelrc** in the root folder of the project  **PRESET CONFIGURATION IN .baberc**  {  "presets": ["@babel/preset-env"]  }   * **test :** This takes a regex value .This property tell webpack that babel loader will be applied to JS file only * **use:** This property makes babel compartible with webpack. * The preset-env module transpiles the ES6 code to ES5 code so that the transpiled code is compartible to lower version of browsers too. |  |

**CONVERTING TO ES6 MODULES SYSTEM**

Let’s convert the above commonJs module system into ES6 module system. Below are few highlight of the difference between them



|  |  |
| --- | --- |
| **Sum.js** | **Index.js** |
| const sum = (a, b) => a + b;  **export default sum;** | **import sum from "./sum";**  const total = sum(10, 10);  console.log(total); |

### ES6 MODULES SYSTEM

Modules can load each other and use special directives **export and import** to interchange functionality, call functions of one module from another one:

* export keyword labels variables and functions that should be accessible from outside the current module.
* import allows to import functionality from other modules.

#### EXPORT

The export statement is used when creating JavaScript modules to export functions, objects, or primitive values from the module so they can be used by other programs with the import statement. There are two types of exports:

1. Named Exports (Zero or more exports per module)
2. Default Exports (One per module)

We can have multiple **named** exports but one and one only **default** export per module.

#### DEFAULT EXPORT

|  |  |
| --- | --- |
| **DEFAULT** CLASS EXPORT (depend.js) | **DEFAULT** CLASS IMPORT (script.js) |
| **VARIATION 1**  class Person {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  getFullName() {  return this.firstName + " " + this.lastName;  }  }  export default Person; | import **per** from "./depend";  class Programmer {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  }  let p1 = new Programmer("Mike", "Clark");  let person = new per();  console.log(person.getFullName.call(p1)); |
| **VARIATION 2**  export default Person {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  getFullName() {  return this.firstName + " " + this.lastName;  }  } | **ADDING SCRIPT TO HTML**  <script **type="module"** src="js/script.js"></script>   1. For export default we use the keyword “export default” 2. In the imported file default exported modules can be imported by any name (i.e “**per**”) |

|  |  |  |
| --- | --- | --- |
| **DEFAULT** FUNCTION EXPORT (depend.js) | | **DEFAULT** FUNCTION IMPORT (script.js) |
| **VARIATION 1**  function sayHello() {  console.log("Hello World!");  }  export default sayHello; | **VARIATION 2**  export default sayHello () {  console.log("Hello World!");  } | import hello from "./depend.js";  hello(); |

|  |  |  |
| --- | --- | --- |
| **DEFAULT** VARIABLE EXPORT (depend.js) | | **DEFAULT** VARIABLE IMPORT (script.js) |
| **VARIATION 1**  const name = ["Alex", "Maria"];  export default name; | **VARIATION 2 IS INVALID**  ~~export default~~ const name=["Alex", "Maria"]; | import n from "./depend.js";  console.log(n); |

#### NAMED EXPORT

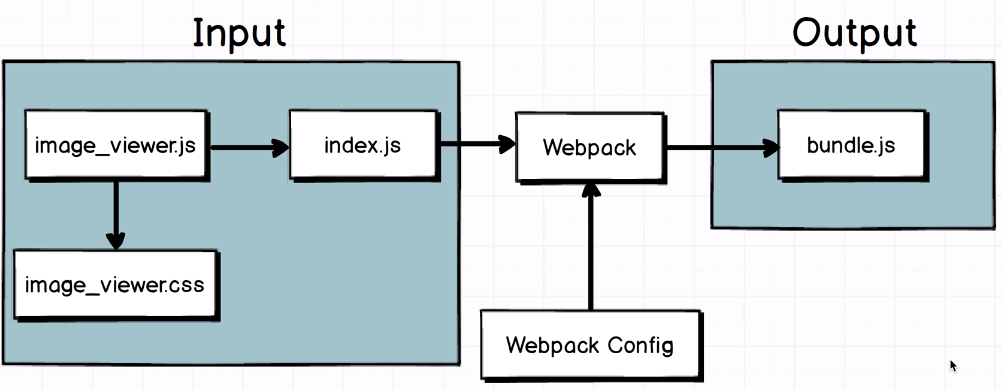
|  |  |
| --- | --- |
| **NAMED** CLASS EXPORT (depend.js) | **NAMED** CLASS IMPORT (script.js) |
| **VARIATION 1**  class Person {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  getFullName() {  return this.firstName + " " + this.lastName;  }  }  export {Person}; | import { Person } from "./depend.js";  class Programmer {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  }  let p1 = new Programmer("Mike", "Clark");  let person = new Person();  console.log(person.getFullName.call(p1)); |
| **VARIATION 2**  export Person {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  getFullName() {  return this.firstName + " " + this.lastName;  }  } | **ADDING SCRIPT TO HTML**  <script **type="module"** src="js/script.js"></script>   1. For export default we use the keyword “export default” 2. In the imported file default exported modules can be imported by any name (i.e “**per**”) |

|  |  |  |
| --- | --- | --- |
| **NAMED** FUNCTION EXPORT (depend.js) | | **NAMED** FUNCTION IMPORT (script.js) |
| **VARIATION 1**  function sayHello() {  console.log("Hello World!");  }  export {sayHello}; | **VARIATION 2**  export sayHello () {  console.log("Hello World!");  } | import { sayHello } from "./depend.js";  sayHello(); |

|  |  |  |
| --- | --- | --- |
| **NAMED** VARIABLE EXPORT (depend.js) | | **NAMED** VARIABLE IMPORT (script.js) |
| **VARIATION 1**  const name = ["Alex", "Maria"];  export {name}; | **VARIATION 2**  export const name = ["Alex", "Maria"]; | import {name} from "./depend.js";  console.log(name); |

|  |  |  |
| --- | --- | --- |
|  | **EXPLICIT IMPORTING** | **IMPORTING USING ALIAS NAME** |
| class Person {  sayName() {  console.log("Name is Alex");  }  }  function sayHello() {  console.log("Hello World");  }  const mobile = 123;  export { Person, sayHello, mobile }; | import { Person, sayHello, mobile } from "./depend.js";  let person = new Person();  console.log(person);  sayHello();  console.log(mobile); | import \* as details from "./depend.js";  let person = new details.Person();  console.log(person);  console.log();  details.sayHello();  console.log(details.mobile); |

### CSS FILES IN WEBPACK



* To bundle the CSS file using webpack we need to couple of css-loader
* To make the CSS file part of the bundled file we need below two css loader

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **INSTALLING CSS LOADER**  **npm install --save-dev css-loader style-loader** | | **PACKAGE.JSON UPDATED**  {  ….  },  …  "devDependencies": {  "@babel/core": "^7.9.6",  "@babel/preset-env": "^7.9.6",  "babel-loader": "^8.1.0",  **"css-loader": "^3.5.3",**  **"style-loader": "^1.2.1",**  "webpack": "^2.7.0"  }  } | | |
| **WEB CONFIG**   1. To plug in the css loaders we need to add a rule. Note that the order of loading of the loader is from right to left 2. The css loader is only applied to css files only (test property) | | const path = require("path");  const config = {  entry: "./src/index.js",  output: {  path: path.resolve(\_\_dirname, "build"),  filename: "bundle.js"  },  module: {  rules: [  {  use: "babel-loader",  test: /\.js$/  },  {  **use: ["style-loader", "css-loader"],**  **test: /\.css$/**  }  ]  }  };  module.exports = config; | | |
|  | **Image\_viewer.css** | **image\_viewer.js** | | **Index.js** |
| img {  border: 10px solid red;  } | import "./image\_viewer.css";  const image = document.createElement("img");  image.src = "https://i.picsum.photos/id/1/400/400.jpg";  document.body.appendChild(image); | | **import sum from "./sum";**  **import "./image\_viewer";**  const total = sum(10, 10);  console.log(total); |
| 1. Lets create two files “image\_view.js” and “image\_viewer.css” 2. Image\_viewer.js file creates an image and append it to DOM. It also imports the css file into it. 3. Finally this file is imported to the index.js(entry file). 4. Note – The way of importing the image\_viewer.js file is different from “sum” module import, because- we don’t need anything from image\_viewer.js- We just want it to exexute when index.js is executed.when we import any file, it actually gets executed when index.js file executes. | | | |
|  | | * The style dynamically added to head section of the HTML document. * This is the JS file which actually dynamically add it in the DOM * ***Note here styles are not loads as a separate entity (we cannot see any network call for CSS)*** | |
| **FLOW DIAGRAM** | | | | |

#### GENERATING THE CSS AS SEPARATE ENTITY

1. To load the css file separately we need to use a plugin “extract-text-webpack-plugin”
2. **INSTALLING THE PLUG-IN** : **npm install --save-dev extract-text-webpack-plugin**

|  |  |  |
| --- | --- | --- |
| **WEB CONFIG JS**  const path = require("path");  **const ExtractTextPlugin = require("extract-text-webpack-plugin");**  const config = {  entry: "./src/index.js",  output: {  path: path.resolve(\_\_dirname, "build"),  filename: "bundle.js"  },  module: {  rules: [  {  use: "babel-loader",  test: /\.js$/  },  **{**  **use: ExtractTextPlugin.extract({**  **loader: "css-loader"**  **}),**  **test: /\.css$/**  **}**  ]  },  **plugin: [new ExtractTextPlugin("style.css")]**  };  module.exports = config; | 1. We use a plug-in here not a loader 2. The bundled css will be created in the **bundle** folder as “**style.css**”   **HTML FILE**  <head>  **<link rel="stylesheet" href="./build/style.css" />**  </head>  <body>  <script src="./build/bundle.js" async defer></script>  </body>  </html> | |
|  | The webpack build process will create a separate css file which can then can be included in HTML |

### CODE SPLITING USING WEBPACK

Webpack usually bundles all the code as on single file . The complete file gets loaded even it not needed on a particult page

1. Code splitting means loading code when it is needed
2. Code splitting can be done using webpack using ”System.import()”

|  |  |
| --- | --- |
|  | Lets consider an example   1. Page will have a “click” button 2. On clicking the button 🡪 it will load the JS file 🡪 which in turn loads the image 3. This whole process is asynchronous |
|  |