

## JS

<https://repl.it/languages/javascript>

```
var shop = {
  getCostPrice: function() {
    return 100;
  },
  getSellingPrice: function() {
    let calculateProfit = function(){
      return this.getCostPrice() * 0.2;
    }
    return this.getCostPrice() + calculateProfit();
  }
}
```

```
console.log(shop.getSellingPrice());
```

**What will be the output of the following?**

```
let y = 1;
if (function f() {}) {
  y += typeof f;
  console.log(y);
}
```

**Output:** 1undefined

**First class functions in JS?****Write function constructors in ES5 and ES6?**

// ES5 Function Constructor

```
function Person(name) {
  this.name = name;
}
```

// ES6 Class

```
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

The main difference in the constructor comes when using inheritance. If we want to create a Student class that subclasses Person and add a studentId field, this is what we have to do in addition to the above.

// ES5 Function Constructor

```
function Student(name, studentId) {
  // Call constructor of superclass to initialize superclass-derived members.
  Person.call(this, name);
```

```
  // Initialize subclass's own members.
  this.studentId = studentId;
}
```

```
Student.prototype = Object.create(Person.prototype);
```

```
Student.prototype.constructor = Student;
```

// ES6 Class

```
class Student extends Person {
  constructor(name, studentId) {
```

```
    super(name);
    this.studentId = studentId;
  }
}
```

---

### IIFEs in js?

**Ans:**

```
(function IIFE(){
    console.log( "Hello!" );
})();
```

---

**console.log(3 < 2 < 1);**

**Output:** true

---

### What will be the output of the following?

```
function buildFunctions() {
  var arr = [];
  for (var i = 0; i < 3; i++) {
    arr.push(
      function() {
        console.log(i);
      }
    )
  }
  return arr;
}
```

```
var fs = buildFunctions();
fs[0]();
fs[1]();
fs[2]();
```

**Output:** 3, 3, 3

---

### What will be the output of the following?

```
function buildFunctions2() {
  var arr = [];
  for (var i = 0; i < 3; i++) {
    arr.push(
      (function(j) {
        return function() {
          console.log(j);
        }
      })(i);
    )
  }
  return arr;
}
```

```
var fs2 = buildFunctions2();
fs2[0]();
fs2[1]();
fs2[2]();
```

**Output:** 0, 1, 2

---

### Call vs Bind vs apply?

#### What will be the output of the following?

```
function multiply(a, b) {
  return a*b;
}
```

```
var multiplyByTwo = multiply.bind(this, 2);
```

```
multiplyByTwo(4);
```

**Output:** 8

**What will be the output of the following?**

```
function multiply(a, b) {  
  return a*b;  
}
```

```
var multiplyByTwo = multiply.bind(this, 2, 2);  
multiplyByTwo(5);
```

**Output:** 4

---

**Insert an item into an array at specific index?**

---

**How to eliminate duplicates in an array?**

---

**Higher Order Functions in JS?**

---

**Difference between debounce and throttle?**

---

**for..in vs for..of?**

---

**for..each vs map?**

---

**Destructuring and Rest/Spread**

```
const obj = {  
  id: 1,  
  status: 'Active'  
}
```

```
test(obj);  
function test() {  
  
}
```

---

**Promises, Async/Await?**

---

**every vs some?**

---

**Axios vs Fetch?**

---

**Did you Interceptors in Axios?**

---

**RxJs**

---

**What is difference between BehaviorSubject and Observable ?**

Observable is stateless  
BehaviourSubject is stateful

Observable creates copy of data  
BehaviourSubject shares data

Observable is unidirectional  
BehaviourSubject is bidirectional

---

**What is difference between Observables and Promises ?**

Promises runs asynchronously and we get the return value only once.  
Observables runs asynchronously and we get the return value multiple times

Promises are not lazy  
Observables are lazy

Promises cannot be cancelled  
Observables can be cancelled

Promises provide a single future value.  
Observables provide multiple future values.

---

### What are the operators in RxJS ?

The operators in RxJS are as follows-

tap()  
catchError()  
switchAll()  
finalize()  
throwError()

---

### Differences between services and factories?

---

#### Output of the following

```
let func = function() {  
  {  
    let l = 'let';  
    let v = 'var';  
  }  
  console.log(v);  
  console.log(l);  
}
```

func();

To solve this use closures, where v will also be undefined and won't be accessed o

```
let func = function() {
```

```
  {  
    (function(){  
      let l = 'let';  
      let v = 'var';  
    })()  
  }  
  console.log(v);  
  console.log(l);  
}
```

func();

//output : l not defined, as l has block scope  
v will be logged, as v has function scope

---

#### Output of the following

console.log(5<6<7); // true

5 < 6 => true , true < 7, true will be converted to 1 and compared which results returning true

console.log(7<6<5); // false

7 < 6 => true, true > 5, as true will be converted to 1 and compared which results returning false

---

```
function abc(){  
  console.log(this);  
}
```

abc();

Ans: **this** gives the current window object when called

---

```
function abc(){
  var a = 10;
  console.log(this.a);
} abc();
Ans: 10
```

---

```
var a = 10;
function abc(){
  console.log(this.a);
  var a = 5;
} abc();
Ans: 10
```

---

```
var a = 10;
function abc() {
  var a = 5;
  console.log(this.a);
} abc();
Ans: 10
```

---

Different phases of javascript event ?

**Event.NONE** - No event is being processed at this time.

**Event.CAPTURING\_PHASE** - The event is being propagated through the target's ancestor objects. [Window](#) -> [Document](#) -> [HTMLHtmlElement](#), and so on through the elements until the target's parent is reached. [Event listeners](#) registered for capture mode when [EventTarget.addEventListener\(\)](#) was called are triggered during this phase.

**Event.AT\_TARGET** - The event has arrived at [the event's target](#). Event listeners registered for this phase are called at this **time**. If [Event.bubbles](#) is false, processing the event is finished after this phase is complete.

**Event.BUBBLING\_PHASE** - The event is propagating back up through the target's ancestors in reverse order, starting with the parent, and eventually reaching the containing [Window](#). This is known as bubbling, and occurs only if [Event.bubbles](#) is true. [Event listeners](#) registered for this phase are triggered during this process.

---

Event propagation/Event bubbling ?

**Event bubbling** directs an event to its intended target, it works like this:

A button is clicked and the event is directed to the button. If an event handler is set for that object, the event is triggered. If no event handler is set for that object, the event bubbles up (like a bubble in water) to the objects parent.

---

Difference between target and currentTarget?

```
<p onclick="">
  <span clicked></span>
</p> - click event bubbles up from <span> to <p>
```

**target** is the element that triggered the event (e.g., the user clicked on <span>)

**currentTarget** is the element that the event listener is attached to. <p>

---

Difference between stopImmediatePropagation and stopPropagation?

**stopPropagation** will prevent any **parent** handlers from being executed.

**stopImmediatePropagation** will do the same **but also** prevent other handlers from executing

**return false;** is equal to **e.preventDefault()** and **e.stopPropagation();**

---

Difference between undefined and null?

**undefined** means that the variable has not been declared, or has not been given a value.

**Ex.** var TestVar; alert(TestVar); //shows undefined

**null** is a special object because typeof null returns 'object'.

**Ex.** var TestVar = null; alert(TestVar); //shows null

---

Web Sockets ?

The [WebSocket](#) specification defines an API establishing "socket" connections between a web browser and a server. In plain words: There is an persistent connection between the client and the server and both parties can start sending data at any time.

---

Web Workers ?

When executing scripts in an HTML page, the page becomes unresponsive until the script is finished.

A web worker is a JavaScript that runs in the background, independently of other scripts, without affecting the performance of the page. You can continue to do whatever you want: clicking, selecting things, etc., while the web worker runs in the background.

---

### Closure ?

A closure is an inner function that has access to the variables in the outer (enclosing) function's scope chain. The closure has access to variables in three scopes; specifically:

- (1) variable in its own scope,
- (2) variables in the enclosing function's scope, and
- (3) global variables.

```
var globalVar = "xyz";
(function outerFunc(outerArg) {
  var outerVar = 'a';
  (function innerFunc(innerArg) {
    var innerVar = 'b';
    console.log(
      "outerArg = " + outerArg + "\n" + "\\123
      "innerArg = " + innerArg + "\n" + "\\456
      "outerVar = " + outerVar + "\n" + "\\a
      "innerVar = " + innerVar + "\n" + "\\b
      "globalVar = " + globalVar + "\\xyz
    );
  })(456);
})(123);
```

---

### Function Hoisting ?

Function hoisting means that functions are moved to the top of their scope.

A function declaration doesn't just hoist the function's name. It also hoists the actual function definition

```
// Outputs: "Yes!"
isItHoisted();
```

```
function isItHoisted() {
  console.log("Yes!");
}
```

However, **function definition hoisting only occurs for function declarations**, not function expressions. For example:

```
// Outputs: "Definition hoisted!"
definitionHoisted();
```

```
// TypeError: undefined is not a function
definitionNotHoisted();
```

```
function definitionHoisted() {
  console.log("Definition hoisted!");
}
```

```
var definitionNotHoisted = function () {
  console.log("Definition not hoisted!");
};
```

So, if you use a named function expression:

```
// ReferenceError: funcName is not defined
funcName();
```

```
// TypeError: undefined is not a function
varName();
var varName = function funcName() {
  console.log("Definition not hoisted!");
};
```

---

### Difference between call(), bind() and apply()

Use .bind() when you want that function to later be called with a certain context, useful in events. Use .call() or .apply() when you want to invoke the function immediately, and modify the context.

Call/apply call the function immediately, whereas bind returns a function that when later executed will have the correct context set for calling the original function. This way you can maintain context in async callbacks, and events.

In call the subsequent arguments are passed in to the function as they are, while apply expects the second argument to be an array that it unpacks as arguments for the called function.

---

### What does Object.create() do?

The **Object.create()** method creates a new object with the specified prototype object and properties.

Object.create() can be used to achieve classical inheritance.

---

### Associative Arrays?

Many programming languages support arrays with named indexes.  
Arrays with named indexes are called **associative arrays** (or hashes).  
JavaScript does not support arrays with named indexes.  
In JavaScript, arrays always use numbered indexes.

---

I have a function with `this.name = ""`, how can I alert my name without passing any params?  
Using prototypes u can call ur name without passing any params to the function

```
function Emp () {
    this.name = "Kapil";
}
Emp.prototype.sayName = function () {
    console.log("Hi, my name is " + this.name);
}
var me = new Emp();
me.sayName();
```

---

ECMA6?

let keyword introduced in place of var  
CONST introduced for global variables

---

How to create a constructor in Javascript?

The constructor property is assigned to function prototype. Then the interpreter forgets about it.  
Additional code is usually required to prevent it from being overwritten.

If you intent to keep it correct, then put it into the new prototype, like this:

```
function Rabbit() { }

Rabbit.prototype = { constructor: Rabbit }

var rabbit = new Rabbit()

alert( rabbit.constructor == Rabbit ) // now fine
```

---

**Objects, Arrays and Array Like Objects** in Javascript?

Using `{}` instead of `new Object()`; is know as “**Object Literal**” syntax.

Javascript **arrays** are a type of object used for storing multiple values in a single variable. Each value gets numeric index and may be any data type.

**Array-like objects** look like arrays. They have various numbered elements and a length property. But that’s where the similarity stops.  
Array-like objects do not have any of Array’s functions, and for-in loops don’t even work! (Ex. `document.forms[0]`)

```
function takesTwoParams(a, b){
    alert(" your parameters were " + arguments.join(", "));
    // throws a type error because arguments.join doesn't exist but arguments[0] gives you a value
}
Using Prototype will help us solve this
function takesTwoParams(a, b){
    var args = Array.prototype.slice.call(arguments);
    alert(" your parameters were " + args.join(", ")); // yay, this works!
}
```

**Array:** This object is the original array that all other arrays inherit their properties from.

**Array.prototype:** This gives us access to all the methods properties that each array inherits

**Array.prototype.slice:** The original slice method that is given to all arrays via the prototype chain. We can’t call it directly though, because when it runs internally, it looks at the **this** keyword, and calling it here would make this point to Array, not our arguments variable.

**Array.prototype.slice.call():** `call()` and `apply()` are prototype methods of the Function object, meaning that they can be called on every function in javascript. These allow you to change what the **this** variable points to inside a given function.

And finally, you get a regular array back! This works because javascript returns a new object of type Array rather than whatever you gave it. This causes a lot of headaches for a [few people](#) who are trying to make subclasses of Array, but it’s very handy in our case!

---

Javascript Prototypes?

All JavaScript objects inherit the properties and methods from their prototype.

Objects created using an object literal, or with `new Object()`, inherit from a prototype called `Object.prototype`.

Objects created with `new Date()` inherit the `Date.prototype`.

The `Object.prototype` is on the top of the prototype chain.

---

Webservice call in plain javascript?

```

<html>
<head>
  <title>SOAP JavaScript Client Test</title>
  <script type="text/javascript">
    function soap() {
      var xmlhttp = new XMLHttpRequest();
      xmlhttp.open('POST', 'https://somesoapurl.com/', true);
      // build SOAP request
      var sr =
        '<?xml version="1.0" encoding="utf-8"?>' + '<soapenv:Envelope ' +
        'xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ' +
        'xmlns:api="http://127.0.0.1/Integrics/Enswitch/API" ' +
        'xmlns:xsd="http://www.w3.org/2001/XMLSchema" ' +
        'xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">' +
        '<soapenv:Body>' +// ... Login - username and password etc ...
        '</soapenv:Body>' +
        '</soapenv:Envelope>';
      xmlhttp.onreadystatechange = function () {
        if (xmlhttp.readyState == 4) {
          if (xmlhttp.status == 200) {
            alert('done. use firebug/console to see network response');
          }
        }
      }
      // Send the POST request
      xmlhttp.setRequestHeader('Content-Type', 'text/xml');
      xmlhttp.send(sr);
    }
  </script>
</head>
<body>
  <form name="Demo" action="" method="post">
    <div><input type="button" value="Soap" onclick="soap();" /></div>
  </form>
</body>
</html>

```

---

## Websockets vs HTTP?

WebSockets provides these benefits over HTTP:

- Persistent stateful connection for the duration of connection

- Low latency: near real-time communication between server/client due to no overhead of reestablishing connections for each request as HTTP requires.

- Full duplex: both server and client can send/receive simultaneously

WebSocket and HTTP protocol have been designed to solve different problems, i.e. WebSocket was designed to improve bi-directional communication whereas HTTP was designed to be stateless, distributed using a request/response model. Other than the sharing the ports for legacy reasons (firewall/proxy penetration), there isn't much of a common ground to combine them into one protocol.

---

## Bubble Sorting?

```
var numbers = [12, 10, 15, 11, 14, 13, 16];
```

```

function bubbleSort(array) {
  var next2last = array.length - 1,
      holder,
      swapOccured,
      index,
      nextIndex;

  array.some(function () {
    swapOccured = false;
    for (index = 0; index < next2last; index += 1) {
      nextIndex = index + 1;
      if (array[index] > array[nextIndex]) {
        holder = array[nextIndex];
        array[nextIndex] = array[index];
        array[index] = holder;
        swapOccured = true;
      }
    }
  })
}

```



```
    }  
    if (!swapOccured) {  
        return true;  
    }  
    return false;  
});  
}  
bubbleSort(numbers);  
console.log(numbers);
```

# HTML/CSS

Friday, November 9, 2018 4:28 PM

## Remove all styling with one line of CSS

```
.button {  
  all: unset; // will remove all the existing/default styles  
}
```

---

## em vs rem

The difference between em and rem units is how the browser determines the px value they translate into.

rem

When using rem units, the pixel size they translate to depends on the font size of the root element of the page, i.e. the html element. That root font size is multiplied by whatever number you're using with your rem unit.

For example, with a root element font size of 16px, 10rem would equate to 160px, i.e.  $10 \times 16 = 160$ .

em

When using em units, the pixel value you end up with is a multiplication of the font size on the element being styled.

They are relative to the font size "of the element on which they are used".

For example, if a div has a font size of 18px, 10em would equate to 180px, i.e.  $10 \times 18 = 180$ .

- Translation of rem units to pixel value is determined by the font size of the html element. This font size is influenced by inheritance from the browser font size setting unless explicitly overridden with a unit not subject to inheritance.

- Any sizing that doesn't need em units for the reasons described above, and that should scale depending on browser font size settings.

Translation of em units to pixel values is determined by the font size of the element they're used on. This font size is influenced by inheritance from parent elements unless explicitly overridden with a unit not subject to inheritance.

---

Different positioning of elements?

## STATIC, RELATIVE, ABSOLUTE, FIXED

Default position of element – **STATIC**

**STATIC** - They are displayed in the page where they rendered as part of normal HTML flow. Statically positioned elements don't obey left, top, right and bottom rules

**RELATIVE** - Relative positioning allows you to specify a specific offset (left, top etc) which is relative to the element's normal position in HTML flow.

**ABSOLUTE** – Absolute positioning whereby you specify the exact location of the element relative to the entire document, **or the next relatively positioned element further up the element tree.**

**FIXED** - Fixed positioning restricts an element to a specific position in the viewport, which stays in place during scroll.

---

Block and Inline elements?

Block-Level

If no width is set, will expand naturally to fill its parent container

Can have margins and/or padding

If no height is set, will expand naturally to fit its child elements (assuming they are not floated or positioned)

Ignores the vertical-align property

**Examples** : <p>, <div>, <form>, <header>, <nav>, <ul>, <li>, and <h1>

Inline

Flows along with text content, thus INLINE

Will not clear previous content to drop to the next line like block elements

Is subject to **white-space** settings in CSS

Will ignore the **width** and **height** properties

Will ignore top and bottom margin settings, but will apply left and right margins, and any padding

Is subject to the vertical-align property

**Examples**: <a>, <span>, <b>, <em>, <i>, <code>, <mark>, and <code>.

---

New in HTML5 when compared to HTML4?

<audio> and <video> tags.

<svg> and <canvas>

JS Geolocation API

App Cache, WebSQL db, Web storage

Web Sockets

<!DOCTYPE html>

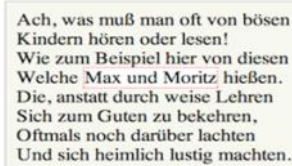
Semantic/Structural elements - <section>, <header>, <footer>, <details>, <summary>, <nav>, <ruby>, <rt>, <time>, <wbr>

---

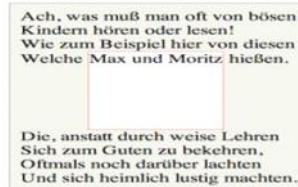
Difference between display:inline, display:inline-block and display:block in css?

- Imagine a <span> element inside a <div>. If you give the <span> element a height of 100px and a red border for example, it will look like this with.

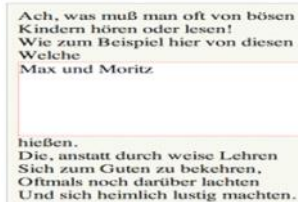
- **display: inline**



- **display: inline-block**



- **display: block**



- Elements with display:inline-block elements are like display:inline elements, but they can have a **width** and **height**. So you can use an inline-block element as a block while flowing it within text or other elements.

---

DataList in HTML5?

The <datalist> tag specifies a list of pre-defined options for an <input> element.

The <datalist> tag is used to provide an "**autocomplete**" feature on <input> elements. Users will see a drop-down list of pre-defined options as they input data.

Use the <input> element's list attribute to bind it together with a <datalist> element.

```
<input list="browsers">

<datalist id="browsers">
  <option value="Internet Explorer">
  <option value="Firefox">
  <option value="Chrome">
  <option value="Opera">
  <option value="Safari">
</datalist>
```

---

Box-sizing?

The box-sizing property is used to tell the browser what the sizing properties (width and height) should include.

box-sizing: content-box|border-box|initial|inherit;

---

CSS3 vs CSS?

Animations

Rounded corners and gradients.

Combinator - New addition of General Sibling Combinator is done to match sibling elements of a given element through tilde (~) Combinator.

Pseudo-Elements - a new convention of double colons '::' has been introduced that allow in-depth yet easy styling.

Background Style Properties - background image, position and repeat.

Box-sizing, Box-shadow, text-shadow

---

# Problem Solving

Wednesday, August 18, 2021 3:26 PM

## PROBLEM SOLVING

---

### firstDuplicate

```
arr = [2,1,3,5,4, 3, 7, 12]; resArr = [arr[0]];
for (let i = 1; i < arr.length; i++) {console.log('i:'+i);
  if(resArr.indexOf(arr[i]) !== -1) { console.log('arr:'+arr[i]); break;}
  resArr.push(arr[i]);
  if (i === arr.length-1) { console.log(-1); }
}
```

---

### Eliminate duplicates in an array using single line of code

JS introduced a new data structure called Set which will remove duplicates

```
let arr = [1, 2, 2, 3];
console.log(new Set(arr)); //output: {1, 2, 3} is a set but not an array

console.log([...new Set(arr)]); //To get the output as an array use a spread operator
```

---

I have a production unit, where there are around 50 machines doing the production. They have been enabled with monitoring and auto correcting systems. There could be cases where the machine could be down for some reason or the other and the auto correction system kicks in to fix the issue. The monitoring system keeps track of the time intervals where the machine was down. It stores the data in [start time, end time] for each machine. I want to do some auditing at the end of month and find out

The total duration of time where my production unit is not functioning to its full capacity

Example data: machine 1 [00-00.15],[3-3.10],[14-14.08],[22-22.30]  
Machine 2 [15-15.30],[22-22.15]

---

Find and print the uncommon characters of the two given strings.

Example data:

deloitte  
dell  
Answer  
oit

```
let strOne = 'deloitte';
let strTwo = 'dell';
let chars = [];
let str = strOne + " " + strTwo;
let obj = {};
for (let i = 0; i < str.length; i++) {
  if (typeof(obj[str.charAt(i)]) !== 'undefined') {
    obj[str.charAt(i)] = obj[str.charAt(i)]+1;
    if (i >= strOne.length) {
      let index = chars.indexOf(str.charAt(i));
      if (index > -1) chars.splice(index, 1);
      console.log(str.charAt(i) + " : "+index);
    }
    continue;
  }
  obj[str.charAt(i)] = 0;
  chars.push(str.charAt(i));
}
console.log(JSON.stringify(obj));
console.log(chars);
let output = chars.length > 0 ? chars.join(" ") : 'No uncommon characters';
console.log(output);
```

-----  
-----

Longest number possible.

Given a number “S” and number of digits “n”, write code a program to find the largest “n” digit number whose sum of digits is equal to “S”

for example if S=20 and n=3, the number is 992.

the program should cover the edge cases like if S=30 and n=3, then there is no possible number similarly for S=2 and n=3

-----  
-----

For a = [2, 4, 7], the output should be absoluteValuesSumMinimization(a) = 4.

for x = 2, the value will be  $\text{abs}(2 - 2) + \text{abs}(4 - 2) + \text{abs}(7 - 2) = 7$ .

for x = 4, the value will be  $\text{abs}(2 - 4) + \text{abs}(4 - 4) + \text{abs}(7 - 4) = 5$ .

for x = 7, the value will be  $\text{abs}(2 - 7) + \text{abs}(4 - 7) + \text{abs}(7 - 7) = 8$ .

The lowest possible value is when x = 4, so the answer is 4.

-----  
-----

```
inputString = "(bar)"
reverseStringInBrackets(inputString) = "rab"
```

```
inputString = "foo(bar)baz"
reverseStringInBrackets(inputString) = "foorabbaz"
```

```
inputString = "foo(bar)baz(blim)"
reverseStringInBrackets(inputString) = "foorabbazmilb"
```

```
inputString = "foo(bar(baz))blim"
reverseStringInBrackets(inputString) = "foobazrabblim"
```

-----  
-----

Sort an array of 0s, 1s and 2s?  
[0, 1, 2, 0, 1, 2] => [0, 0, 1, 1, 2, 2]

-----  
-----

For s1 = "aabcc" and s2 = "adcaa", the output should be  
countOfCommonCharacter(s1, s2) = 3

-----  
-----

For a = [-1, 150, 190, 170, -1, -1, 160, 180], the output should be  
heightSort(a) = [-1, 150, 160, 170, -1, -1, 180, 190].

-----  
-----

Merge sort using js?

-----  
-----

Find out the first Duplicate in array?

- For a = [2, 1, 3, 5, 3, 2], the output should be firstDuplicate(a) = 3.  
There are 2 duplicates: numbers 2 and 3. The second occurrence of 3 has a smaller index than the second occurrence of 2 does, so the answer is 3.
- For a = [2, 2], the output should be firstDuplicate(a) = 2;
- For a = [2, 4, 3, 5, 1], the output should be firstDuplicate(a) = -1.

-----  
-----

Replace alphabets in string with next alphabet

Given a string, your task is to replace each of its characters by the next one in the English alphabet; i.e. replace a with b, replace b with c, etc (z would be replaced by a).  
Example

For `inputString = "crazy"`, the output should be `alphabeticShift(inputString) = "dsbaz"`.

-----  
-----

# React/Redux

Wednesday, August 18, 2021 3:31 PM

## React Playground

<https://jscomplete.com/playground>

<https://jsfiddle.net/reactjs/69z2wepo/>

**JSX?** Syntactical sugar to `React.createElement`

**Libraries required for React App?**

**Function components vs Class components? When to use what?**

**Function components**

**Can we update the Props? If yes, how?**

Props are immutable

**Hooks in React?**

Hooks are functions that let you “hook into” React state and lifecycle features from function components.

**useState** - to preserve this state between re-renders

**useEffect** - Similar to `componentDidMount`, `componentDidUpdate` in Class Components

Lesser used Hooks

**useContext** -

**useReducer** - `useReducer` is usually preferable to `useState` when you have complex state logic that involves multiple sub-values or when the next state depends on the previous one. `useReducer` also lets you optimize performance for components that trigger deep updates because you can pass dispatch down instead of callbacks.

**React Fragments?**

**Did you create any custom hook in React?**

**Reconciliation (Diffing Algorithm)** - API to not worry about what changes on every update.

React uses diffing algorithm to predict component updates for high performance using `key` prop for list items and difference between elements checking against Virtual DOM.

**Container vs Presentational Components?**

**What is Redux and Why Redux?**

- 1 immutable store
- Actions trigger changes
- Reducers updates the state

**Explain the Redux flow using an example?(Add/Edit/Delete a Todo item)**

**What is Immutability? What are immutable in React?**

**Context API?**

**Refs in React?**



### **Form Validations using React?**

### **Higher Order Components in React?**

A higher-order component (HOC) is an advanced technique in React for reusing component logic.

### **Redux vs Flux?**

### **Different ways to pass data from component to component?**

- Lift State
- React Context
- Redux

### **Compilation process difference between Angular and React?**

Angular - JIT/AOT

React - JSX transpilation

### **Do you know about SSR? Why SSR?**

### **Does React support SSR? Frameworks for SSR?**

Server side rendering

React can be SSR'ed using Next.js and Gatsby

Adv:

- SEO
- Performance

# Node Js

Wednesday, August 18, 2021 3:32 PM

## **Node Js**

### **Talk about Event Emitters in NodeJS?**

### **Talk about Event Loops in NodeJS?**

#### **readFileSync vs readSync?**

- readFileSync will make the code synchronous and waits for the file to be read and till that it is loaded it won't execute the next line of code.
- readFile is asynchronous and other code can be run parallelly. It takes a callback with err and data as arguments to run the callback function once the file is loaded.

#### **Where will be the buffer content stored?**

Buffer content is not stored in memory, it is stored in heap memory of V8

#### **What are pipes used for in NodeJS?**

#### **Give an example for fully buffered file access and partially buffered file access?**

fs.createReadStream - Partially buffered

fs.readFile - Fully Buffered

#### **How do you upload files in Node js http server?**

#### **What features does Express middleware provide?**

#### **Tools you used to debug Node apps?**

nodemon

#### **Did you use Websockets in NodeJS?**

# TS

Wednesday, August 18, 2021 3:32 PM

Interfaces

modules

class private properties

# Angular 8

Wednesday, August 18, 2021 3:33 PM

## ANGULAR

### - Entry Components in Angular

- An entry component in Angular is any component that is loaded by its class, not selector.
- Entry components must be registered inside the app or feature module, under `entryComponents` section.
- But this does not apply to all components, Angular automatically registers some entry components for you.
- For instance, any component found in your routes will be registered by Angular CLI during compilation. Therefore, most developer deal with entry components without even realizing so.
- Why to register `entryComponents`?
  - Normally, all components are declared inside the app or [feature module](#) under declarations. This applies to all components inside your project, whether they have been used or not. This is even before you consider components declared by third party libraries that you might not need in your project.
  - If all these unused components were to end up in the final app bundle, they would considerably increase its size. And remember, our goal is to have as small final bundle as possible. To work around this, Angular CLI tree shakes your app during build time. It removes all components that have not been referenced (declared) in your template.
  - Now, remember that an entry component is used or referenced by its class and not inside some other components template (at least most of the time). Without a list of entry components, Angular CLI would fail to include these components in the final bundle, hence breaking your application. Therefore, for most entry components, you need to register them under `entryComponents` array, in your app or [feature module](#).

```
...
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    ...
  ],
  providers: [],
  bootstrap: [AppComponent],
  entryComponents: [
    SomeEntryComponent
  ]
})
export class AppModule { }
```

### - Angular Server-side rendering

## Server-side rendering (SSR) with Angular Universal

This guide describes Angular Universal, a technology that renders Angular applications on the server.

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.

For a more detailed look at different techniques and concepts surrounding SSR, please check out this [article](#).

You can easily prepare an app for server-side rendering using the [Angular CLI](#). The CLI schematic `@nguniversal/express-engine` performs the required steps, as described below.

In this example, the Angular CLI compiles and bundles the Universal version of the app with the [Ahead-of-Time \(AOT\) compiler](#). A Node Express web server compiles HTML pages with Universal based on client requests.

- To create the server-side app module, `app.server.module.ts`, run the following CLI command.

```
ng add @nguniversal/express-engine
```

The command creates the following folder structure.

```
src/
  index.html           app web page
  main.ts              bootstrapper for client app
  main.server.ts       * bootstrapper for server app
  style.css            styles for the app
  app/ ...             application code
  app.server.module.ts * server-side application module
  server.ts            * express web server
  tsconfig.json        TypeScript client configuration
  tsconfig.app.json    TypeScript client configuration
  tsconfig.server.json * TypeScript server configuration
  tsconfig.spec.json   TypeScript spec configuration
  package.json         npm configuration
```

The files marked with \* are new and not in the original tutorial sample.

## Universal in action

To start rendering your app with Universal on your local system, use the following command.

```
npm run build:ssr && npm run serve:ssr
```

## Why use server-side rendering?

There are three main reasons to create a Universal version of your app.

1. Facilitate web crawlers through [search engine optimization \(SEO\)](#)
2. Improve performance on mobile and low-powered devices
3. Show the first page quickly with a [first-contentful paint \(FCP\)](#)

### Facilitate web crawlers (SEO)

Google, Bing, Facebook, Twitter, and other social media sites rely on web crawlers to index your application content and make that content searchable on the web. These web crawlers may be unable to navigate and index your highly interactive Angular application as a human user could do.

Angular Universal can generate a static version of your app that is easily searchable, linkable, and navigable without JavaScript. Universal also makes a site preview available since each URL returns a fully rendered page.

## Improve performance on mobile and low-powered devices

Some devices don't support JavaScript or execute JavaScript so poorly that the user experience is

## Improve performance on mobile and low-powered devices

Some devices don't support JavaScript or execute JavaScript so poorly that the user experience is unacceptable. For these cases, you may require a server-rendered, no-JavaScript version of the app. This version, however limited, may be the only practical alternative for people who otherwise couldn't use the app at all.

## Show the first page quickly

- Displaying the first page quickly can be critical for user engagement. Pages that load faster perform better, [even with changes as small as 100ms](#). Your app may have to launch faster to engage these users before they decide to do something else.

With Angular Universal, you can generate landing pages for the app that look like the complete app. The pages are pure HTML, and can display even if JavaScript is disabled. The pages don't handle browser events, but they *do* support navigation through the site using `routerLink`.

In practice, you'll serve a static version of the landing page to hold the user's attention. At the same time, you'll load the full Angular app behind it. The user perceives near-instant performance from the landing page and gets the full interactive experience after the full app loads.

### - zone.js in Angular app

## 1. Why Should I Care About Zone.js?

Angular introduced `Zone.js` to handle change detection. This allows Angular to decide when the UI has to be refreshed. Usually, you don't have to care about any of this, because Zone.js just works.

○

However, if something goes wrong with Zone.js it can be very frustrating to analyze and understand. This is why every developer should know some basics about Zone.js.

## 2. In a Nutshell: How Does Zone.js Work?

Zone.js patches all common async APIs like `setTimeout`, `setInterval`, the promise API, etc. to keep track of all async operations.

Here are the basic concepts you should understand:

### ○ Zone

Zone is a mechanism for intercepting and keeping track of asynchronous work.

### Tasks

For each async operation, Zone.js creates a task. A task is run in one zone.

## NgZone

By default, in an Angular app every task runs in the “Angular” Zone, which is called `NgZone`. There is only one Angular Zone and change detection is triggered exclusively for async operations which run in the `NgZone`.

○

### Root Zone/Forks

Zone.js zones are hierarchical which means you always start with a top-level Zone — the “root” Zone. New Zones can be created by forking the root Zone. `NgZone` is also a fork of the root Zone.

## ZoneSpecs

When forking a Zone, a new Zone will be created based on a `ZoneSpec`. A

- `ZoneSpec` can just include a name for the new child Zone, or can include various Hook methods which can be used to intercept certain Zone/task events.

## 3. You Don't Have to Use Zone.js With Angular (but You Probably Should)

- Zone.js can easily be deactivated during bootstrapping of an Angular application by passing a `noop` Zone. However, you give up change detection, which means you have to decide yourself when the UI has to be refreshed (e.g. via `ChangeDetectorRef.detectChanges()`).

Let's do an experiment. Go and take your favorite Angular 2 app, and *don't* include zone.js. See what happens, I'll wait here. Notice anything interesting? None of the bindings are working!

- Take a look at some of the [code](#). The Application subscribes to zone's `onTurnDone` event, which, [if we do some more digging](#), calls `tick`, which actually calls all of the change detectors. So, without zones, we don't get any change detection, so we don't get any of the nice UI updates that we'd expect!

*Note: It's here that I should say that zones are not required for change detection in general, they simply are the means by which Angular picks up changes and then calls tick so that any listeners for those changes are actually fired. Thanks Maxim Koretskyi!*

○

Why does any of this matter? Well, in Angular 2, we don't have a `$digest` like in the original angular, so if we were to do something like this in Angular 2:

```

let marker = new google.maps.Marker({
  position: new google.maps.LatLng(1, 1),
  map: someMap,
  title: 'Title of a marker'
});

marker.addListener('click', () => {
  this.someProperty = Math.random();
});

```

Then the UI wouldn't actually update properly, because someProperty actually is updated outside of Angular's zone. So we have to include something like this in order to get the correct updates:

```

marker.addListener('click', () => {
  this.zone.run(() => {
    this.someProperty = Math.random();
  });
});

```

In other words, zone.run() is kind of like the new \$digest().

#### - Handling multiple subscriptions of observables in Angular

```

import { Observable, of, interval } from 'rxjs';
import { delay } from 'rxjs/operators';

export function getSingleValueObservable() {
  return of('single value');
}

export function getDelayedValueObservable() {
  return of('delayed value').pipe(delay(2000));
}

```

```

export function getMultiValueObservable() {
  return new Observable<number>(observer => {
    let count = 0;
    const interval = setInterval(() => {
      observer.next(count++);
      console.log('interval fired');
    }, 1000);

    return () => {
      clearInterval(interval);
    };
  });
}

```

#### ○ ForkJoin



```
import { Component, OnInit } from '@angular/core';
import { forkJoin } from 'rxjs';
import { map } from 'rxjs/operators';

import {
  getSingleValueObservable,
  getDelayedValueObservable,
  getMultiValueObservable
} from '../util';

@Component({
  selector: 'app-fork-join-operator',
  templateUrl: './fork-join-operator.component.html'
})
```

```
export class ForkJoinOperatorComponent {
  show = false;
  values$ = forkJoin(
    getSingleValueObservable(),
    getDelayedValueObservable()
    // getMultiValueObservable(), forkJoin on works for observables that complete
  ).pipe(
    map(([first, second]) => {
      // forkJoin returns an array of values, here we map those values to an object
      return { first, second };
    })
  );
}
```

In our component, we use `forkJoin` to combine the Observables into a single value Observable. The `forkJoin` operator will subscribe to each Observable passed into it. Once it receives a value from all the Observables, it will emit a new value with the combined values of each Observable. `ForkJoin` works well for single value Observables like Angular `HttpClient`. `ForkJoin` can be comparable to `Promise.All()`.

```
<h2>forkJoin Operator</h2>

<button (click)="show = !show">Toggle</button>

<ng-container *ngIf="show">
  <ng-container *ngIf="values$ | async; let values;">
    <p>{{values.first}}</p>
    <p>{{values.second}}</p>
    <p>{{values.third}}</p>
  </ng-container>
</ng-container>
```

If you take note with the `ForkJoin`, we don't get any values rendered until all Observables pass back a value. In our working demo, we don't see any values until the delayed Observable emits its value. This technique allows us to clean up our template by subscribing once letting Angular handle our subscription while referencing our Observable multiple times. The issue we see here is that `forkJoin` limits us to single value Observables only. The next example we will see how to handle multi-value Observables.

- **combineLatest**

```
import { Component, OnInit } from '@angular/core';
import { combineLatest } from 'rxjs';
import { map } from 'rxjs/operators';

import {
  getSingleValueObservable,
  getDelayedValueObservable,
  getMultiValueObservable
} from '../util';

@Component({
  selector: 'app-combine-latest-operator',
  templateUrl: './combine-latest-operator.component.html'
})
```

```
export class CombineLatestOperatorComponent {
  show = false;
  values$ = combineLatest(
    getSingleValueObservable(),
    getDelayedValueObservable(),
    getMultiValueObservable()
  ).pipe(
    map(([first, second, third]) => {
      // combineLatest returns an array of values, here we map those values to an object
      return { first, second, third };
    })
  );
}
```

As we can see `combineLatest` is very similar to `forkJoin`. Combine Latest will emit the single combined value as soon as at least every Observable emits a single value and will continue emitting multiple values with the latest of each.

```
<h2>combineLatest Operator</h2>

<button (click)="show = !show">Toggle</button>

<ng-container *ngIf="show">
  <ng-container *ngIf="values$ | async; let values;">
    <p>{{values.first}}</p>
    <p>{{values.second}}</p>
    <p>{{values.third}}</p>
  </ng-container>
</ng-container>
```

- **Subscribing with async pipes**

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

import {
  getSingleValueObservable,
  getDelayedValueObservable,
  getMultiValueObservable
} from '../util';

@Component({
  selector: 'app-async-pipe-object',
  templateUrl: './async-pipe-object.component.html'
})
export class AsyncPipeObjectComponent {
  show = false;
  first$ = getSingleValueObservable();
  second$ = getDelayedValueObservable();
  third$ = getMultiValueObservable();
}
```

```
<h2>Async Pipe Object</h2>
<button (click)="show = !show">Toggle</button>

<ng-container *ngIf="show">
  <ng-container
    *ngIf="{ first: first$ | async, second: second$ | async, third: third$ | async } as values;"
  >
    <p>{{values.first}}</p>
    <p>{{values.second}}</p>
    <p>multi values {{values.third}}</p>
  </ng-container>
</ng-container>
```

#### - Life cycle hooks in Angular

```
-- Lifecycle Hook Log --

#1 name is not known at construction
#2 OnChanges: name initialized to "Windstorm"
#3 OnInit
#4 DoCheck
#5 AfterContentInit
#6 AfterContentChecked
#7 AfterViewInit
#8 AfterViewChecked
#9 DoCheck
#10 AfterContentChecked
#11 AfterViewChecked
#12 DoCheck
#13 AfterContentChecked
#14 AfterViewChecked
#15 OnDestroy
```

#### - Interceptors

- We need to import `HttpInterceptor` from `'@angular/common/http'`
- We have an `intercept` method through which we can set the common headers for all the requests
- This intercept code will run before request leaves the app
- `intercept` accepts 2 arguments
  - req of type `HttpRequest` which needs to be imported from `'@angular/common/http'` - request obj of any data it might yield
  - next of type `HttpHandler` which also needs to be imported from `'@angular/common/http'` - obj which forwards the request to continue the journey

- ◆ It has handle method to continue the request

```

1 import { HttpInterceptor, HttpRequest, HttpHandler } from '@angular/common/http';
2
3 export class AuthInterceptorService implements HttpInterceptor {
4   intercept(req: HttpRequest<any>, next: HttpHandler) {
5     console.log('Request is on its way');
6     return next.handle(req);
7   }
8 }

```

```

3 import { BrowserModule } from '@angular/platform-browser';
4 import { HttpClientModule, HTTP_INTERCEPTORS } from '@angular/common/http';
5
6 import { AppComponent } from './app.component';
7 import { AuthInterceptorService } from './auth-interceptor.service';
8
9 @NgModule({
10   declarations: [AppComponent],
11   imports: [BrowserModule, FormsModule, HttpClientModule],
12   providers: [
13     {
14       provide: HTTP_INTERCEPTORS,
15       useClass: AuthInterceptorService,
16       multi: true
17     }
18   ],
19   bootstrap: [AppComponent]
20 })
21 export class AppModule {}
22

```

- In app.module.ts, we need to mention the HTTP\_INTERCEPTORS in the providers array though an object to make sure all the requests go through this interceptor
  - provide: HTTP\_INTERCEPTORS -> Needs to be imported from '@angular/http/common'
  - useClass: Class name of our interceptor
  - multi: true -> To tell we have more than one interceptors
- We can modify the request obj before it has to be sent
  - We cannot directly do that, but we can clone the request object and modify it

```

1 import {
2   HttpInterceptor,
3   HttpRequest,
4   HttpHandler
5 } from '@angular/common/http';
6
7 export class AuthInterceptorService implements HttpInterceptor {
8   intercept(req: HttpRequest<any>, next: HttpHandler) {
9     console.log('Request is on its way');
10    const modifiedRequest = req.clone({
11      headers: req.headers.append('Auth', 'xyz')
12    });
13    return next.handle(modifiedRequest);
14  }
15 }
16

```

- We can also manipulate the data received through the request using pipe operator through the tap method
  - tap can be imported from 'rxjs/operators' to look into the response.
  - We get an event in the interceptor through which we can detect the type whether response or request

```

6  } from '@angular/common/http';
7  import { tap } from 'rxjs/operators';
8
9  export class AuthInterceptorService implements HttpInterceptor {
10   intercept(req: HttpRequest<any>, next: HttpHandler) {
11     console.log('Request is on its way');
12     console.log(req.url);
13     const modifiedRequest = req.clone({
14       headers: req.headers.append('Auth', 'xyz')
15     });
16     return next.handle(modifiedRequest).pipe(
17       tap(event => {
18         console.log(event);
19         if (event.type === HttpEventType.Response) {
20           console.log('Response arrived, body data: ');
21           console.log(event.body);
22         }
23       })
24     );
25   }
26 }

```

□ We can also use map to transform the response data

- We can also use multiple interceptors

```

4  import { HttpClientModule, HTTP_INTERCEPTORS } from '@angular/common/http';
5
6  import { AppComponent } from './app.component';
7  import { AuthInterceptorService } from './auth-interceptor.service';
8  import { LoggingInterceptorService } from './logging-interceptor.service';
9
10 @NgModule({
11   declarations: [AppComponent],
12   imports: [BrowserModule, FormsModule, HttpClientModule],
13   providers: [
14     {
15       provide: HTTP_INTERCEPTORS,
16       useClass: AuthInterceptorService,
17       multi: true
18     },
19     {
20       provide: HTTP_INTERCEPTORS,
21       useClass: LoggingInterceptorService,
22       multi: true
23     }
24   ]
25 })

```

- Using angular component in react

- Ionic page life cycle methods

- viewWillEnter
- viewDidEnter
- viewWillLeave
- viewDidLeave

- every

## Definition and Usage

The `every()` method checks if all elements in an array pass a test (provided as a function).

The `every()` method executes the function once for each element present in the array:

- - If it finds an array element where the function returns a *false* value, `every()` returns false (and does not check the remaining values)
  - If no false occur, `every()` returns true

**Note:** `every()` does not execute the function for array elements without values.

**Note:** `every()` does not change the original array



```

<p>Click the button to check if every element in the array has a value
of 18 or more.</p>

<button onclick="myFunction()">Try it</button>

<p id="demo"></p>

<script>
var ages = [32, 33, 16, 40];

○ function checkAdult(age) {
  return age >= 18;
}

function myFunction() {
  document.getElementById("demo").innerHTML = ages.every(checkAdult);
}
</script>

</body>
</html>

```

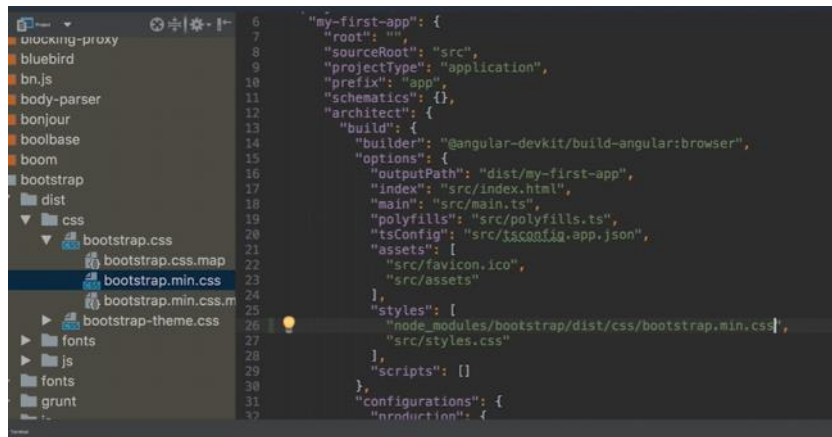
Click the button to check if every element in the array has a value of 18 or more.

Try it

false

- Have you used any third party theming libraries in Angular like bootstrap? If yes, how?

- To install bootstrap
  - npm install --save-dev bootstrap@3



- How do you implement two-way binding in Angular?

- Two-way binding

Important: For Two-Way-Binding (covered in the next lecture) to work, you need to enable the `ngModel` directive. This is done by adding the `FormsModule` to the `imports[]` array in the `AppModule`.

- You then also need to add the import from `@angular/forms` in the `app.module.ts` file:

```
import { FormsModule } from '@angular/forms';
```

- Directives in Angular?

- Two types of built-in directives are available in Angular
  - Attribute Directives (1-way)
    - `[ngClass]="{'color-text: true', 'font-size': true}"` // Pass as an object to pass multiple classes
  - Structural Directives (2-way) - Changes the DOM

- \*ngFor, \*ngIf, [ngSwitch] .. \* ngSwitchcase
- Use trackBy in \*ngFor for increasing the performance so that the entire DOM of the component does not get re-rendered

## - How to create a Directive in Angular?

- Creating directive via cli
  - ◆ ng g directive <directive-name>
- Using Renderer to build a directive
  - ◆ We need to import Renderer2 from '@angular/core'
  - ◆ Renderer2 is changed to Renderer
  - ◆ This is a better way to access the DOM elements and set style via renderer methods
  - ◆ Renderer has an in-built setStyle method to set the style we want to the element by accessing the element via ElementRef

```

1 import { Directive, ElementRef, Renderer2 } from '@angular/core';
2
3 @Directive({
4   selector: '[appBetterHighlight]'
5 })
6 export class BetterHighlightDirective implements OnInit {
7   constructor(private elRef: ElementRef, private renderer: Renderer2) {}
8
9   ngOnInit() {
10    this.renderer.setStyle(this.elRef.nativeElement, 'background-color', 'blue')
11  }
12 }
  
```

- But the above approach is not that interactive, so to listen to all the events to the element to which the directive is attached to, we need to use @HostListener

- ◆ @HostListener decorator is imported from '@angular/core'
- ◆ The event that needs to be triggered is added as an argument as a string inside @HostListener

```

1 import { Directive, ElementRef, Renderer2, HostListener } from '@angular/core';
2
3 @Directive({
4   selector: '[appBetterHighlight]'
5 })
6 export class BetterHighlightDirective implements OnInit {
7   constructor(private elRef: ElementRef, private renderer: Renderer2) {}
8
9   ngOnInit() {
10    // this.renderer.setStyle(this.elRef.nativeElement, 'background-color', 'blue', false, false);
11  }
12
13   @HostListener('mouseenter') mouseover(eventData: Event) {
14     this.renderer.setStyle(this.elRef.nativeElement, 'background-color', 'blue', false, false);
15   }
16
17   @HostListener('mouseleave') mouseleave(eventData: Event) {
18     this.renderer.setStyle(this.elRef.nativeElement, 'background-color', 'transparent', false, false);
19   }
20 }
  
```

- In order to replace the renderer, we can use @HostBinding

- ◆ @HostBinding decorator need to be imported from '@angular/core'
- ◆ The property of the element that needs to be updated will be added as an argument as a string inside @HostBinding

```

1 import { Directive, ElementRef, HostListener, HostBinding } from '@angular/core';
2
3 @Directive({
4   selector: '[appBetterHighlight]'
5 })
6 export class BetterHighlightDirective implements OnInit {
7   @HostBinding('style.backgroundColor') backgroundColor: string = 'transparent';
8   constructor(private elRef: ElementRef, private renderer: Renderer2) {}
9
10  ngOnInit() {
11    // this.renderer.setStyle(this.elRef.nativeElement, 'background-color', 'blue', false, false);
12  }
13
14   @HostListener('mouseenter') mouseover(eventData: Event) {
15     // this.renderer.setStyle(this.elRef.nativeElement, 'background-color', 'blue', false, false);
16     this.backgroundColor = 'blue';
17   }
18
19   @HostListener('mouseleave') mouseleave(eventData: Event) {
20     // this.renderer.setStyle(this.elRef.nativeElement, 'background-color', 'transparent', false, false);
21     this.backgroundColor = 'transparent';
22   }
23 }
  
```

## - How to create a service and use it only for a particular component?

## - How to create Routes in Angular?

- ◆ We can also load modules related to a route

```

-UI > src > app > TS app-routing.module.ts
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

const routes: Routes = [
  { path: '', redirectTo: 'app', pathMatch: 'full' },
  {
    path: 'app',
    loadChildren: () => import('./main/main.module').then(m => m.MainModule)
  },
  {
    path: 'user',
    loadChildren: () => import('./user/user.module').then(m => m.UserModule)
  },
  {
    path: 'account',
    loadChildren: () => import('./account/account.module').then(m => m.AccountModule)
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }

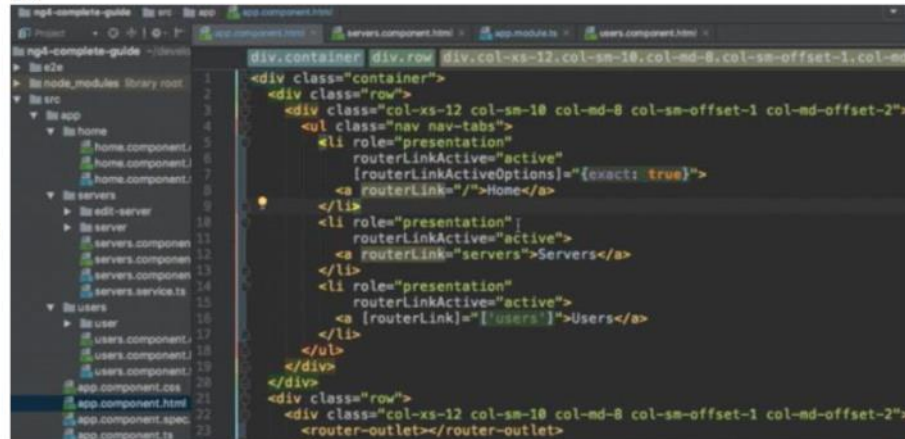
```

## - How to style active router links?

### □ Styling Active Router Links

- ◆ routerLinkActive="<class-name>" will allow us to specify the class for making the link active
- ◆ routerLinkActive analyzes your currently loaded path and then checks which links lead to a route which uses this path and by default, it marks an element as active, it marks this CSS class if it contains the path you are on or if this link is part of the path which is currently loaded.
- ◆ routerLinkActiveOptions is an attribute directive in which we can send an object to pass on the information to make the link active based on the options

◇ Ex.



```


-


```

- ◇ Here as shown in the above screenshot, we pass a Javascript object and that would not work without having our square brackets.
- ◇ So with square brackets, we can pass anything which will be resolved dynamically, like this Javascript object

## - How to fetch route params from route?

- Use ActivatedRoute



```

import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } 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 route: ActivatedRoute) {}

  ngOnInit() {}
}

```

To fetch the param from the url we need to code as in below screenshot

```

import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } 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 route: ActivatedRoute) {}

  ngOnInit() {
    this.user = {
      id: this.route.snapshot.params['id'],
      name: this.route.snapshot.params['name']
    };
  }
}

```

## - View Encapsulation in Angular?

- Styles applied to a component will be applied only to the component but not the child component because of the Encapsulation that Angular provides.
- Angular enforces this style encapsulation.
- Angular emulates the Shadow DOM. even though Shadow DOM is a technology which is not supported by all the browsers where each element has its kind of its own Shadow DOM assigned and its styles will be applied.
- As all browsers do not support this Shadow DOM, as explained below Angular emulates this behavior of View Encapsulation.

We can override this Encapsulation as shown below

- Using the encapsulation property in the @Component decorator metadata

```

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

@Component({
  selector: 'app-server-element',
  templateUrl: './server-element.component.html',
  styleUrls: ['./server-element.component.css'],
  encapsulation: ViewEncapsulation.None
})
export class ServerElementComponent implements OnInit {
  @Input('srvElement') element: string;

  constructor() {}

  ngOnInit() {}
}

```

- ViewEncapsulation needs to be imported from '@angular/core'

- ViewEncapsulation has three modes

- ◆ Emulated (Default) // Applies View Encapsulation
- ◆ None // Removes all the dynamic attributes added to the elements in the component. View Encapsulation will be removed
- ◆ Native // Uses Shadow DOM technology, but gives us the same result as Emulated option but it is not supported by all the browsers

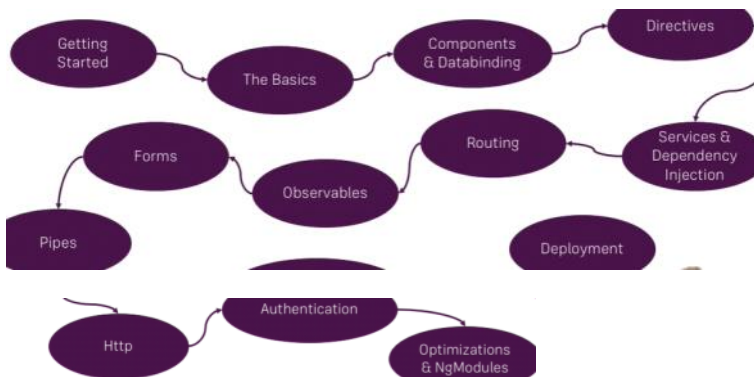
## - Angular Component Lifecycle?

- Component LifeCycle

## Lifecycle

	ngOnChanges	Called after a bound input property changes
	ngOnInit	Called once the component is initialized
	ngDoCheck	Called during every change detection run
○	ngAfterContentInit	Called after content (ng-content) has been projected into view
▪	ngAfterContentChecked	Called every time the projected content has been checked
	ngAfterViewInit	Called after the component's view (and child views) has been initialized
	ngAfterViewChecked	Called every time the view (and child views) have been checked
	ngOnDestroy	Called once the component is about to be destroyed

- How do you implement dynamic forms in Angular?



- What are the two types of compilation in Angular? What is the default compilation type and their difference?

- AOT vs JIT

- Observables vs Subject

- A **Subject** is both an observer and observable. A **BehaviorSubject** a **Subject** that can emit the current value (**Subjects** have no concept of current value). That is the confusing part. The easy part is using it. The **BehaviorSubject** holds the value that needs to be shared with other components.
- An RxJS **Subject** is a special type of **Observable** that allows values to be multicasted to many Observers. While plain **Observables** are unicast (each subscribed Observer owns an independent execution of the **Observable**), **Subjects** are multicast. A **Subject** is like an **Observable**, but can multicast to many Observers.
- **Subject** and **Observable** is that a **Subject** has state, it keeps a list of observers. On the other hand, an **Observable** is really just a function that sets up observation. ... This means a **subject** can be **used** as an observer to subscribe to any **observable**.

- How do you unsubscribe to an observable?

- Entry components in Angular

- An entry component in Angular is any component that is loaded by its class, not selector.
- Entry components must be registered inside the app or feature module, under entryComponents section.
- But this does not apply to all components, Angular automatically registers some entry

- components for you.
- For instance, any component found in your routes will be registered by Angular CLI during compilation. Therefore, most developers deal with entry components without even realizing so.
- Why to register entryComponents?
  - Normally, all components are declared inside the app or [feature module](#) under declarations. This applies to all components inside your project, whether they have been used or not. This is even before you consider components declared by third party libraries that you might not need in your project.
  - If all these unused components were to end up in the final app bundle, they would considerably increase its size. And remember, our goal is to have as small final bundle as possible. To work around this, Angular CLI tree shakes your app during build time. It removes all components that have not been referenced (declared) in your template.
  - Now, remember that an entry component is used or referenced by its class and not inside some other components template (at least most of the time). Without a list of entry components, Angular CLI would fail to include these components in the final bundle, hence breaking your application. Therefore, for most entry components, you need to register them under entryComponents array, in your app or [feature module](#).

```
...
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    ...
  ],
  providers: [],
  bootstrap: [AppComponent],
  entryComponents: [
    SomeEntryComponent
  ]
})
export class AppModule { }
```

## - Angular Server-side rendering

### Server-side rendering (SSR) with Angular Universal

This guide describes **Angular Universal**, a technology that renders Angular applications on the server.

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.

For a more detailed look at different techniques and concepts surrounding SSR, please check out this [article](#).

You can easily prepare an app for server-side rendering using the [Angular CLI](#). The CLI schematic `@nguniversal/express-engine` performs the required steps, as described below.

In this example, the Angular CLI compiles and bundles the Universal version of the app with the [Ahead-of-Time \(AOT\) compiler](#). A Node Express web server compiles HTML pages with Universal based on client requests.

- To create the server-side app module, `app.server.module.ts`, run the following CLI command.

```
ng add @nguniversal/express-engine
```

The command creates the following folder structure.

```
src/
  index.html          app web page
  main.ts             bootstrapper for client app
  main.server.ts      * bootstrapper for server app
  style.css           styles for the app
  app/ ...            application code
    app.server.module.ts * server-side application module
  server.ts           * express web server
  tsconfig.json       TypeScript client configuration
  tsconfig.app.json   TypeScript client configuration
  tsconfig.server.json * TypeScript server configuration
  tsconfig.spec.json  TypeScript spec configuration
  package.json        npm configuration
```

The files marked with \* are new and not in the original tutorial sample.

## Universal in action

To start rendering your app with Universal on your local system, use the following command.

```
npm run build:ssr && npm run serve:ssr
```

## Why use server-side rendering?

There are three main reasons to create a Universal version of your app.

1. Facilitate web crawlers through [search engine optimization \(SEO\)](#)
2. Improve performance on mobile and low-powered devices
3. Show the first page quickly with a [first-contentful paint \(FCP\)](#)

### Facilitate web crawlers (SEO)

Google, Bing, Facebook, Twitter, and other social media sites rely on web crawlers to index your application content and make that content searchable on the web. These web crawlers may be unable to navigate and index your highly interactive Angular application as a human user could do.

Angular Universal can generate a static version of your app that is easily searchable, linkable, and navigable without JavaScript. Universal also makes a site preview available since each URL returns a fully rendered page.

## Improve performance on mobile and low-powered devices

Some devices don't support JavaScript or execute JavaScript so poorly that the user experience is unacceptable. For these cases, you may require a server-rendered, no-JavaScript version of the app. This version, however limited, may be the only practical alternative for people who otherwise couldn't use the app at all.

### Show the first page quickly

- Displaying the first page quickly can be critical for user engagement. Pages that load faster perform better, [even with changes as small as 100ms](#). Your app may have to launch faster to engage these users before they decide to do something else.

With Angular Universal, you can generate landing pages for the app that look like the complete app. The pages are pure HTML, and can display even if JavaScript is disabled. The pages don't handle browser events, but they *do* support navigation through the site using `routerLink`.

In practice, you'll serve a static version of the landing page to hold the user's attention. At the same time, you'll load the full Angular app behind it. The user perceives near-instant performance from the landing page and gets the full interactive experience after the full app loads.

# Angular - latest

Wednesday, August 18, 2021 3:34 PM

does hoisting work in arrow function?

Is ng-content initialized on onInit ?

<ng-content> does not "produce" content, it simply projects existing content. Think of it as some variation of node.appendChild(el) or the well-known JQuery version \$(node).append(el): with these methods, the node is not cloned, it is simply moved to its new location. Because of this the lifecycle of the projected content is bound to where it is declared, not where it is displayed.

There are two reasons for this behavior: consistency of expectations and performance. What "consistency of expectations" means is that as a developer, I can read my app's code and guess its behavior based on the code that I have written

## The solution

To let the wrapper control the instantiation of its children, we need to give it a template for the content, rather than the content itself. This can be done in two ways: using the <ng-template> element around our content, or using a structural directive with the "star" syntax, like \*myContent. For simplicity, we will use the <ng-template> syntax in our examples, but you can find all the information you need about the star prefix [here](https://medium.com/claritydesignsystem/ng-content-the-hidden-docs-96a29d70d11b). Our new app looks like this:

From <<https://medium.com/claritydesignsystem/ng-content-the-hidden-docs-96a29d70d11b>>

What parameters Interceptor has

interceptor method (req, next:handler)  
handle.next(req) after modification

Diff between Promises and Observables

Diff between Component and Directive

ngTemplate and ngContainer

Types of binding 1way, 2way

Module Injector

What is Dependency Injection. And something related to dependency heirarchy

2nd round

how would u do dependency injection in es5

if u had to design angular how would u do it

solid principles, IOC containers

Route Resolvers in Angular

<https://dzone.com/articles/understanding-angular-route-resolvers-by-example>

Does hoisting work on arrow function

Is ng-content initialized on onInit

Promises vs Observables

Life Cycle

Component vs Directive

Different types of promise methods

ngTemplate, ngContainer, ngContent

Module Injector

Lazy Loading

IOC containers, SOLID

ViewChild and ViewChildren

CanActivate, CanDeactivate

Route Resolver

Semantic vs Non-Semantic

Pure and Impure Pipe

Subject vs Behavioral Subject



# Angular Js

Wednesday, August 18, 2021 3:45 PM

Built-in **services** in AngularJS?

`$window` – browser window

`$document` – `window.document`

`$location` – `window.location`

`$log` - To write error, info, warning and debugging into the browser's console

`$parse` - To convert AngularJS expression into a function

`$timeout` - `window.setTimeout` function

`$interval` - `window.setInterval` function.

Built-in **filters** in AngularJS?

`Currency` – To format number as currency.({{amount | currency:"INR":2}})

`Filter` – To filter or search an item from array.({{subjects | filter:subjectName}})

`Lowercase` – To convert any string to lower case({{student.fullName() | lowercase}})

`Uppercase`- To convert any string to upper case({{student.fullName() | uppercase}})

`Number` - To format any number as string

`OrderBy` - To set array order ascending or descending.({{subject in subjects | orderBy:'marks'}})

HTML DOM **directives** in AngularJS?

`ng-disabled`

`ng-show`

`ng-hide`

`ng-click`

**Service vs Factory?**

**Service**

```
app.service('MyService', function () {
    this.sayHello = function () {
        console.log('hello');
    };
});
```

**Factory**

```
app.factory('MyService', function () {
    return {
        sayHello: function () {
            console.log('hello');
        };
    }
});
```

**Service** is a constructor function whereas **Factory** is not.

A **factory** function is just a function that gets called, which is why we have to return an object explicitly whereas in **Service** there's this code that calls **Object.create()** with the service constructor function, when it gets instantiated.

In a **factory** it returns some object and we can run some code before returning that object whereas in **service** it is not possible to run some code before returning the object.

A **service** is a constructor function which can return anything so we can also write a service which returns an object just like the **factory** does as below.

```
app.service('MyService', function () {
```



```
// we could do additional work here too
return {
  sayHello: function () {
    console.log('hello');
  };
};
});
```

**Service** allows us to use ES6 classes

```
class MyService {
  sayHello() {
    console.log('hello');
  }
}
app.service('MyService', MyService);
```

Variable Hoisting in Javascript?

```
// Outputs: undefined
console.log(declaredLater);

var declaredLater = "Now it's defined!";

// Outputs: "Now it's defined!"
console.log(declaredLater);
```

JavaScript treats variables which will be declared later on in a function differently than variables that are not declared at all. Basically, the JavaScript interpreter "looks ahead" to find all the variable declarations and "hoists" them to the top of the function. Which means that the example above is equivalent to this:

```
var declaredLater;

// Outputs: undefined
console.log(declaredLater);

declaredLater = "Now it's defined!";

// Outputs: "Now it's defined!"
console.log(declaredLater);
```

**\$routeProvider vs \$stateProvider?**

### **\$routeProvider**

URLs to controllers and views (HTML partials). It watches \$location.url() and tries to map the path to an existing route definition.

#### **HTML**

```
<div ng-view></div>
```

Above tag will render the template from the \$routeProvider.when() condition which you had mentioned in .config (configuration phase) of angular

#### **Limitations:-**

The page can only contain single ng-view on page

If you SPA has multiple part on page which you need to render on some condition, In this \$routeProvider fails.(In such cases we need to go for directive like ng-include, ng-switch, ng-if, ng-show actually which looks bad as thinking of SPA)  
You cannot relate to routes with each other like who is parent who is child.

### \$stateProvider

- AngularUI Router is a routing framework for AngularJS, which allows you to organize the parts of your interface into a state machine. UI-Router is organized around states, which may optionally have routes, as well as other behavior, attached.

#### **Multiple & Named Views**

Another great feature is the ability to have multiple ui-views view per template.

While multiple parallel views are a powerful feature, you'll often be able to manage your interfaces more effectively by nesting your views, and pairing those views with nested states.

#### **HTML**

```
<div ui-view>
  <div ui-view='header'></div>
  <div ui-view='content'></div>
  <div ui-view='footer'></div>
</div>
```

The majority of ui-router's power is it can manage nested state & views.

#### **Pros**

You can have multiple ui-view on single page

Various view can be nested in each other and maintain by defining state in routing phase.

We can have child & parent relationship here, simply like inheritance in state, also you could define sibling states.

You could change the ui-view="some" of state just by using absolute routing using @ with state name.

Other way you could relative routing using only @ to change ui-view="some", This will replace the ui-view rather than checking it is nested or not.

Here you could use ui-sref to create a href URL dynamically on the basis of URL mentioned in a state, also you could give a state params in the json format.

# Ionic/Cordova

Wednesday, August 18, 2021

3:46 PM

Ionic life cycle hooks?

## Cordova commands

```
npm install -g cordova
```

```
cordova create hello com.example.hello HelloWorld
```

[By default whitelist plugin is added](#)

```
cordova-plugin-whitelist
```

```
cordova-plugin-device
```

```
cordova-plugin-console
```

```
cordova-plugin-network-information
```

```
cordova-plugin-dialogs
```

```
cordova-plugin-statusbar
```

[latest cordova version: 6.3.0](#)

[latest android cordova version: 5.1.0](#)

[latest android ios version: 4.0.1](#)

```
cordova platform add ios --save
```

```
cordova platform add android --save
```

To check if you satisfy requirements for building the platform:

```
cordova requirements
```

```
cordova build
```

```
cordova build ios
```

To add a plugin

```
cordova plugin add cordova-plugin-camera
```

```
cordova platform update android --save
```

```
cordova platform update ios --save
```