Minimum width

This property defines the minimum width of an element. If the content is smaller then the minimum width will be applied.

Margins

Margin Top: It is the space between the top border of an element and the top of the containing block.

Margin Right: It is the space between the right border of an element and the right of the containing block.

Margin Bottom: It is the space between the bottom border of an element and the bottom of the containing block.

Margin Left: It is the space between the top border of an element and the left of the containing block.

Shorthand-

margin: 10px 20px 30px 40px;

1 value -> It applies the same margin all 4 sides.

2 values -> The first margin applies to top and bottom while second to the left and right.

3 values -> First margin applies to the top, second to the right and left and third to the bottom.

4 values -> Margins apply in the clockwise direction i.e. top, right, bottom, left.

Padding

Area between content and border. By default, it is 0.

padding-top: it is the space between top of an element and its top border.

padding-right: it is the space between right of an element and its right border.

padding-bottom: it is the space between bottom of an element and its bottom border.

padding-left: it is the space between left of an element and its left border.

padding-left: 10px;

Shorthand-

padding: 100px;

padding: 10px 20px 30px 40px;

1 value -> It applies the same padding all 4 sides.

2 values -> The first padding applies to top and bottom while second to the left and right.

3 values -> First padding applies to the top, second to the right and left and third to the bottom.

4 values -> Paddings apply in the clockwise direction i.e. top, right, bottom, left.

Border

border: border-width, border-style and border-color values in exact order. These values are applied to all sides of the border.

border-[side]: border-width, border-style and border-color values in exact order. These values are applied to a specified side of the border.

border-width: It takes values / keywords (Thin, Medium or Thick)

border-style: It takes keywords like dashed, solid, double, etc.

border-color: Specified color of the border.

border-[side]-width, border-[side]-style, border-[side]-color: Width, style, color would be applied to specified side.

TASK! Play with border, margins

Background

It is the shorthand notation for applying properties like background color, background image, background repeat, background attachment and background position values.

1. background-color: A color value or keyword that specifies the color of an element's background.

Ex. background-color: yellow;

2. background-image: URL that points to an image.

Ex. background-image: url(https://xyz.com/pic.jpg);

3. background-repeat: A keyword that specifies if and how an image is repeated. Values can be repeat, repeat-x, repeat-y and no-repeat.

Ex. background-repeat: no-repeat;

4. background-attachment: A keyword that specifies whether an image scrolls with the document or remains fixed. Values can be scroll or fixed.

5. background-position: It is used to specify the initial, horizontal and vertical positions of image. The default is top-left.

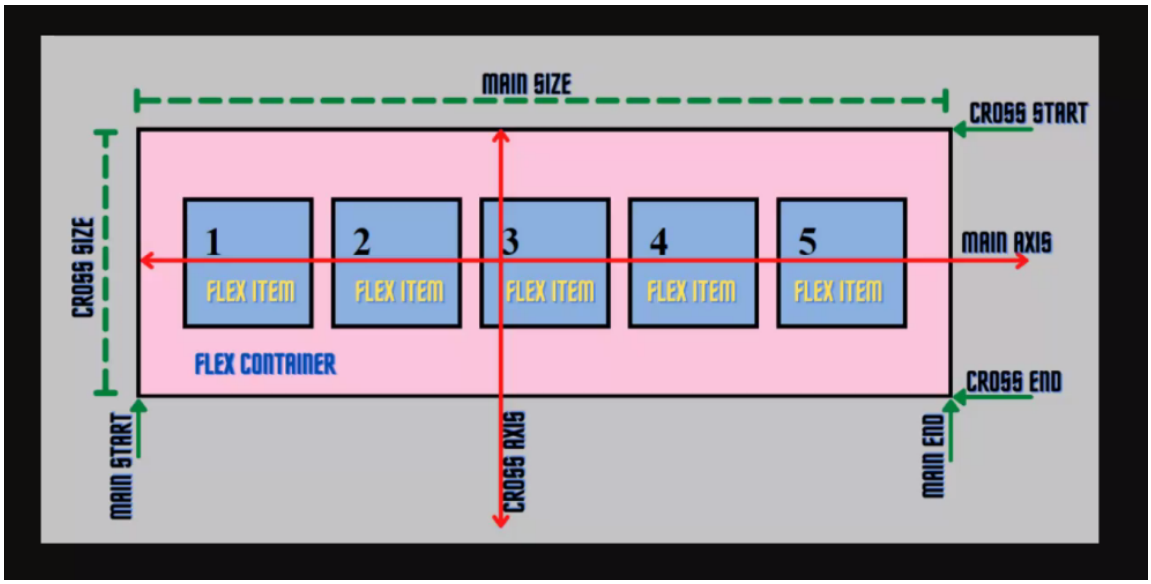
background-size: In this property helps us to set the size of the background image.

Ex. background-size: cover; background-size: 500px 500px;

CSS FlexBox :

flex+Grid very important

Flexbox - 2017



Main axis of a flex container is the primary axis along which the flex items are laid .
Main start - main end : The flex items are laid out within the container starting from mainstart and going to main end .
Cross axis : the axis perpendicular to the main axis is called the cross axis .its direction depends on the main axis direction .

Properties for flex/parent container :

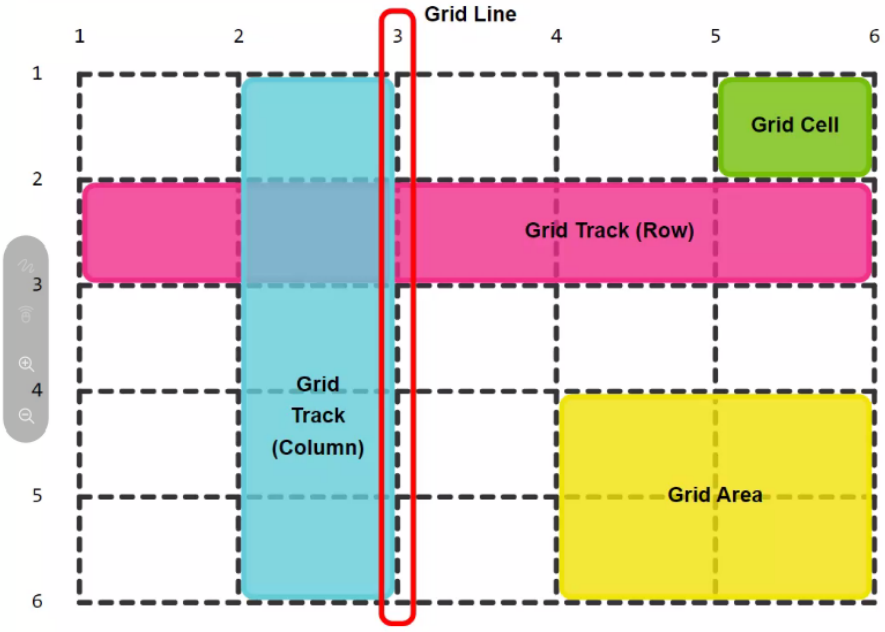
- display : when we assign the value flex to the display property of a container . it turns into a flex container .
- flex-direction : this helps in establishing the main axis direction, thus defining the direction flex items are placed in the flex container .
    1. row : this is the default property used to lay items in the left to right direction
    2. row-reverse : right-left
    3. column : it lays items in top - bottom direction .
    4. column-reverse : bottom-top
- flex-wrap : it allows us to wrap the items as per the need- i.e in a single/multiple lines .
    1. nowrap:(default) it specifies that all the flex items will be laid out in a single line
    2. wrap: flex items will wrap onto multiple lines .
    3. wrap-reverse : it allows flex items to wrap onto multiple lines from bottom to top .
- flex-flow : this is a shorthand for flex direction and flexwrap properties
- Justify-content : it defines how items are positioned along the main axis .
    1. flex-start(default) : flex items are positioned at the beginning of the container .

2. Flex-end : flex items are positioned at the end of the container
3. Start:
4. Center:items are centered along the line .
5. space-between: items are evenly distributed in the line ; first item is on the start line , last item is on the end line .
6. Space-around:items are evenly distributed with equal space around them
7. Space-evenly:items are distributed in such a way that the spacing between any two items (and the space to the edges is equal)

- align-items : it defines how flex items will be laid along the cross axis .
    1. stretch (default) : the items will stretch to fill the container .
    2. flex-start:Items are aligned across the cross start line
    3. Flex-end: items are placed at the cross-end line .
    4. Center:items are centered in the cross axis.
    5. Baseline:items are aligned in a line such that their baselines align
- align-content: When there is extra space in the cross axis, the align-content aligns multiple lines within the container. It is similar to justify-content.
    a. Flex-start
    b. Flex-end
    c. Space-between
    d. Space-around

- gap: It helps us to control the spacing between the items.
- grow: It defines the ability for a flex item to grow if necessary.

GRID :
The CSS grid layout is a 2-D grid based layout system.



- Grid Line: The dividing lines that makesup the structure of the grid. They can be either vertical i.e along column grid lines or horizontal i.e along row grid lines.
- Grid Track: It is the space between two adjacent grid lines.
- Grid Area: The total space surrounded by 4 grid lines/
- Grid item: It is the children of the grid container.
- Grid cell: It is the space between 2 adjacent row and 2 adjacent column grid lines.

Grid Properties :

- Display: It helps us to create a grid container by setting it to the value grid
  Like display:grid
- grid-template-rows
- grid-template-columns
  Defines the columns and rows of the grid with a space separated list of values.

- fr unit: It is the fraction of the free space in the grid.

- grid-gaps(gutters) : grid-row-gap and grid-col : They are used to create gutters between the columns and rows (outer space isn't considered .)

  gap -> shorthand .

bootstrap- > is 12 grid system

Aligning the grid items

- Justify-items: it aligns the grid items along the row axis
    1. Start : it aligns items to the start of the row axis.
    2. End:end of the row axis .
    3. center : align items in the center of cell
    4. stretch: it fills the whole width of the cell
- align-items : Aligns the grid items along the column axis .
    a. stretch(default) : fills the whole height of the cell .
    b. Start:start of the column axis .
    c. end: end of the column axis .
    d. center : align items in the center of the cell

aligning grid tracks relative to the grid container along the row and column axis.

- Justify-content : aligns the grid along the row axis
    1. start: aligns the grid with starting edge of the grid container
    2. end: aligns columns at the end of row axis
    3. center: columns are aligned at the center of the row axis .
    4. space-around : the remaining space of grid container is distributed and applied to the start and end of each column track
    5. Space-between
    6. space-evenly
- Align-content : aligns grid along column axis
    1. start
    2. end
    3. center
    4. stretch
    5. space-around
    6. space-between
    7. Space-evenly

Media Queries :

RWD**:** responsive web design .
Media query helps us to target the browser by certain characteristics and apply styling accordingly.

Responsiveness to certain elements in html . To support different devices .

31-01-2022

From 10:30AM IST

JavaScript

Introduction to JavaScript

How to include JavaScript in HTML

In head section: (Embedded style)

<head>

<script src="src/index.js"></script>

</head>

In body section:

<body>

<script src="src/index.js"></script>

</body>

Syntax

JavaScript is a case sensitive language which is similar to Java.

Creating Identifiers

(Identifier- Name of the variable)

- Identifiers can only contain letters, numbers, underscore (_) and dollar sign ($).
- Identifiers can't start with a number.
- Identifiers are case-sensitive.
- Identifiers can't be the same as reserved words.

Naming recommendations

· Use meaningful names.

· Be consistent. (Use camel case or underscore)

· If using underscore notation, use lowercase for all letters.

· If using camel case, accept the first word, all other words start with capital letters.

Comments

· //Single line comment

· /*

Multiple

Line

comment

*/

Data types

1. Number: Represents an integer or a decimal that can start with a positive or a negative sign.

2. String: Represents character/ string data. It begins and ends with either a single quote or a double quote.

3. Boolean: Represents a Boolean value i.e., true or false.

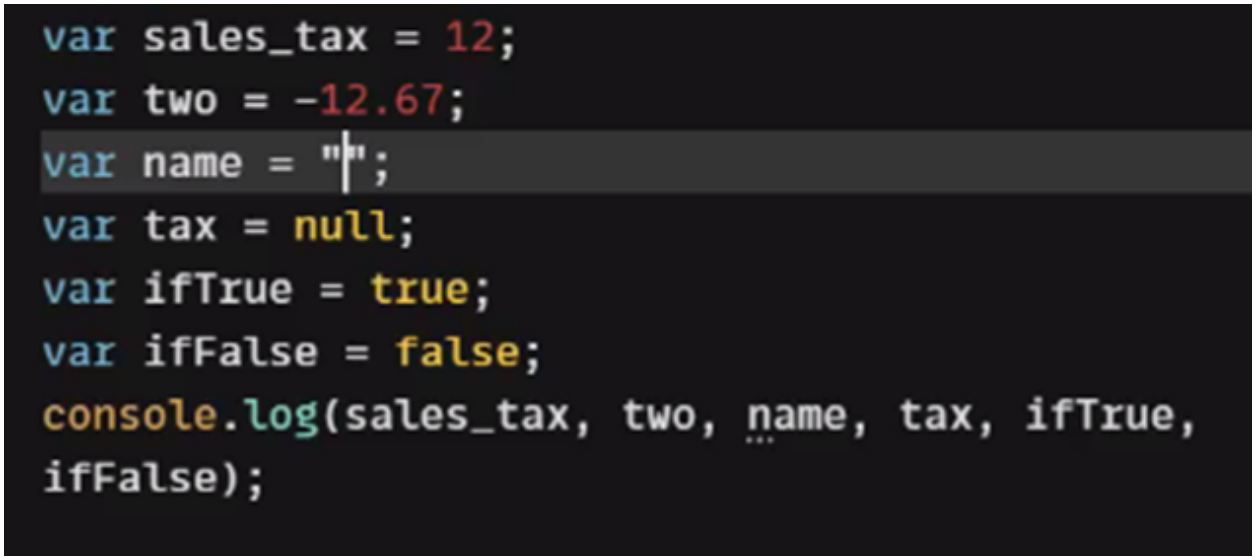
Undefined: Undefined is the value which is assigned to a variable if it's declared but not initialized.

4. Null: This value denoted nothing.

5. Nan: It stands for not-a-number.

// Use ctrl+shift+i to go to the console log.

Variables



Arithmetic Expressions

Normal arithmetic expressions…

Shorthand: sales_tax+=1;

Control Flow statements

1. if
2. if-else
3. if-else-if
4. ternary
5. switch
6. while
7. do-while
8. for loop

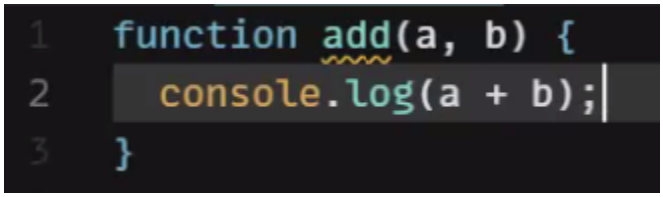
Dom manipulation .

Functions

Skipped

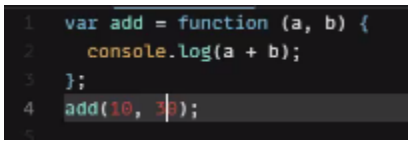
Function Declaration

It is the traditional way to define the function, we start with declaring with the keyword **function** followed by the function name then the parameters.

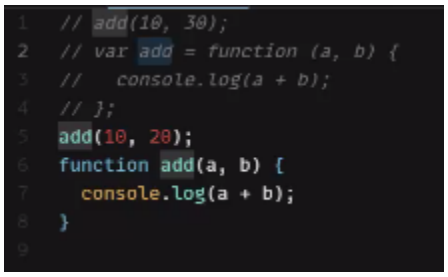


Function expression(anonymous)

We define a function using a variable and store the returned value in that variable.



In function expression, function can’t be called before declaring it.



(Commented code will generate error, uncommented code will output 30)

Function hoisting

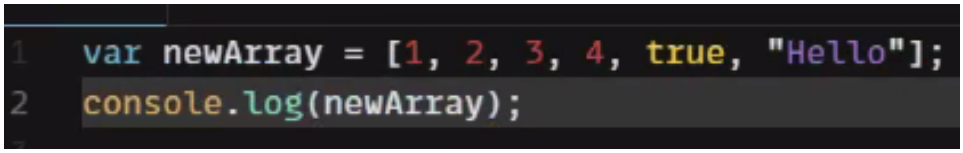
It is a mechanism used by JavaScript engine in which it moves the **function declarations** to the top of the code before executing them.

Immediately Invoked function

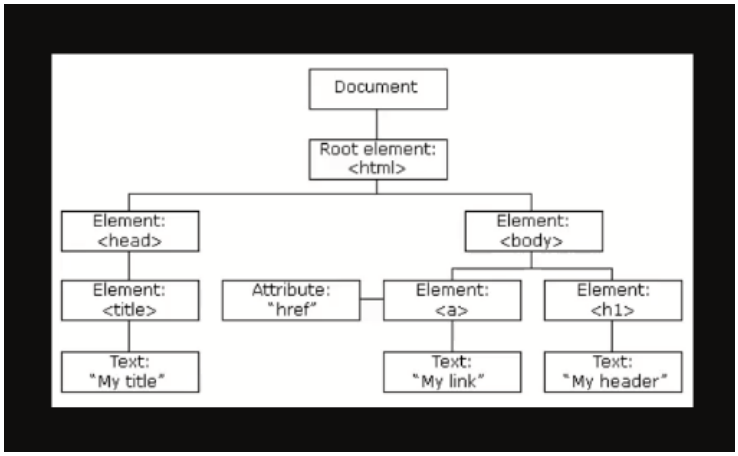
These functions are executed immediately after the declaration.

Q . write function to predict odd and even numbers , prime - 20min

Array



_Document object model :The document object represents the html document that is displayed in that window .

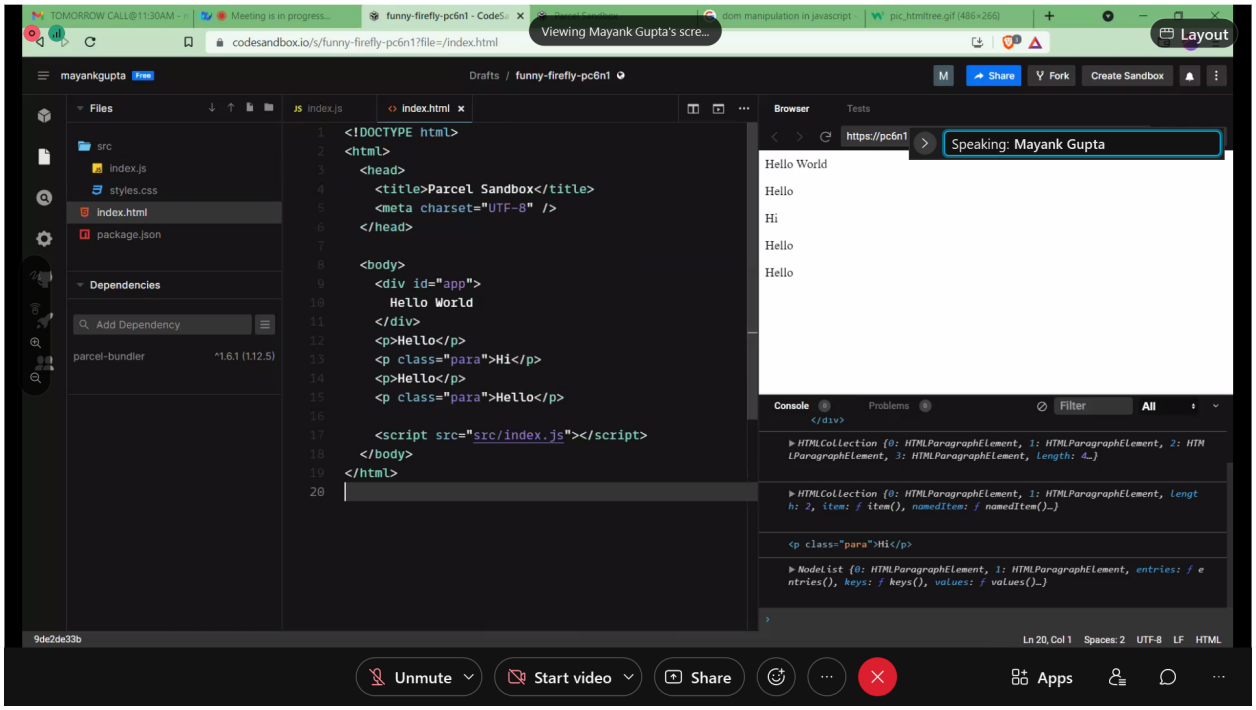


Document object : when an html document is loaded into a window .it becomes a document object

Window object : window object represents an open window , it is the object of the browser

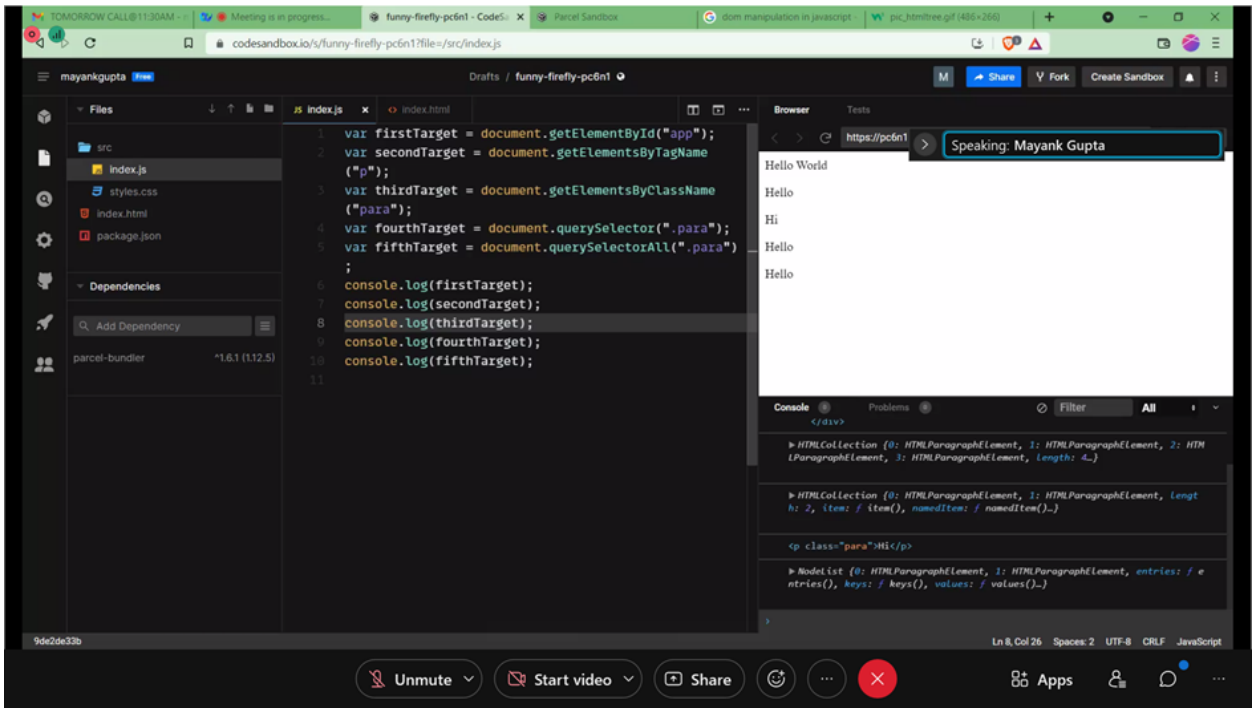
Document methods- Selecting the elements in HTML document

- getElementById = selects an element by id .
- getElementsByTagName = (<p> tag tag) . selects elements by tag name .
- getElementsByClassName = select elements by one or more class names .
- querySelector = select elements by css selectors
- querySelectorAll = it returns a collection of all the elements that matches the css selectors


Viewing Mayank Gupta's scre...
Layout
codesandbox.io/s/funny-firefly-pc6n1?file=/index.html
mayankgupta
Drafts / funny-firefly-pc6n1
Share
Fork
Create Sandbox
Files
index.js
index.html
src
index.js
styles.css
index.html
package.json
Dependencies
Add Dependency
parcel-bundler
^1.6.1 (1.12.5)
<!DOCTYPE html>
<html>
<head>
<title>Parcel Sandbox</title>
<meta charset="UTF-8" />
</head>
<body>

Hello World

Hello
Hi
Hello
Hello
<script src="src/index.js"></script>
</body>
</html>
Browser
Tests
https://pc6n1
Speaking: Mayank Gupta
Hello World
Hello
Hi
Hello
Hello
Console
Problems
Filter
All
9de2de33b
Ln 20, Col 1
Spaces: 2
UTF-8
LF
HTML
Unmute
Start video
Share
Apps


codesandbox.io/s/funny-firefly-pc6n1?file=/src/index.js
mayankgupta
Drafts / funny-firefly-pc6n1
Share
Fork
Create Sandbox
Files
index.js
index.html
src
index.js
styles.css
index.html
package.json
Dependencies
Add Dependency
parcel-bundler
^1.6.1 (1.12.5)
var firstTarget = document.getElementById("app");
var secondTarget = document.getElementsByTagName("p");
var thirdTarget = document.getElementsByClassName("para");
var fourthTarget = document.querySelector(".para");
var fifthTarget = document.querySelectorAll(".para");
console.log(firstTarget);
console.log(secondTarget);
console.log(thirdTarget);
console.log(fourthTarget);
console.log(fifthTarget);
Browser
Tests
https://pc6n1
Speaking: Mayank Gupta
Hello World
Hello
Hi
Hello
Hello
Console
Problems
Filter
All
9de2de33b
Ln 8, Col 26
Spaces: 2
UTF-8
CRLF
JavaScript
Unmute
Start video
Share
Apps

Traversing through the elements :
Parent node : parent node is an html anchor element object in the dom .this is an element node.
Child node : these are the elements that are direct children .

Properties of doc element (this is not method , its a property **)**

- **ParentNode** : getting the parent element
- **firstChild** = Getting the firstchild of the specified parent .

The hash text in the console log is present a text node is inserted to maintain the white space which are there for

- **firstElementChild**
- **lastElementChild**
- **children**
- **nextElementSibling** : Getting the siblings of element

Manipulating elements :

- **createElements** = helps create new element
- **appendChild** = it helps to append a node to a list of child nodes of a specified parent node
- **textContent**(property) : it helps us to get and set the text content of a node .
- **innerHTML** :to get and set the HTML contents of an element .
- **insertBefore**: this method helps to set a new node before an existing node as a child node of a specified parent node .
- **insertAfter** :
- **replaceChild** : this method helps to replace a child element by a new element .
- **removeChild** : this method helps to remove child elements of a node .

Nothing happened before 11AM. We had a small task that's all for 40mins

11am -----> 1 Feb

Working with attributes :
Modifier of an HTML element -> eg . href inside <a>

setAttribute = this method is used to set a value of an attribute on a specified element .
getAttribute = this method is used to get the value of an attribute on a specified element.
removeAttribute = method is used to remove an attribute from a specified element .

hasAttribute = method is used to check if an element has a specified attribute or not .

classList = add a class to an element.
Example: parElem.classList.add("newClass");

Event Handling :

An event is an action that occurs in the web browser, which inturn is used to create changes in a web page

Eg : clickEvent

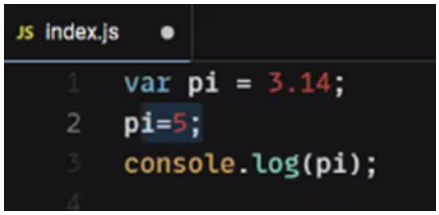
Example : `btn.addEventListener('mouseover', callbackFn())`

Events:

Click = when the mouse is clicked .
Mouseover = when the cursor of the mouse comes over the element .
mouseout = when the cursor of the mouse leaves an element .
mousedown = when the mouse btn is pressed over the element .
mouseup = when mouse btn is released over the element.
keydown, **keyUp** = user press and release the key.
loadevent = when browser finishes the loading of the page .

Let v/s Var v/s Const

Var – Var declared variables are scoped locally and globally. Var declared variables can be re-declared and updated.



Let – Variables declared by the **let** keyword are block scoped. Where block is the area between the curly braces. **Let** can be updated but not re-declared.

Const – It is used to maintain constant value. **Const** declarations are block scoped. **Const** cannot be updated or re-declared. **Const** variables must be initialized at the time of the declaration.

Arrow Functions

```
JS index.js ×
1  let add = (a, b) => {
2    return a + b;
3  };
4  console.log(add(5, 2));
```

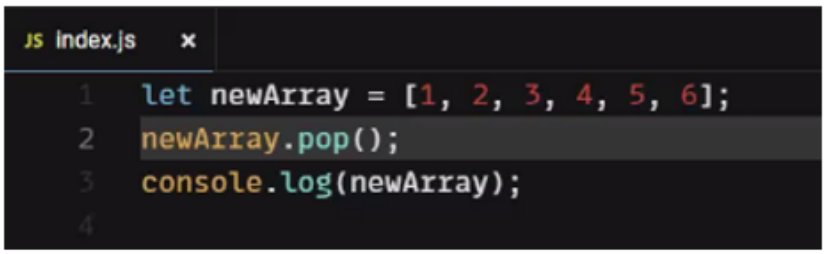
Outputs 7

```
JS index.js ×
1  let add = (a, b) => a + b;
2  console.log(add(5, 2));
3
```

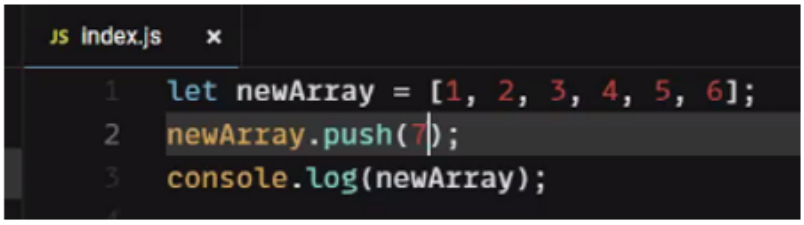
Arrays

```
JS index.js ×
1  let newArray = [1, 2, true, "Hello"];
2  console.log(newArray);
3
```

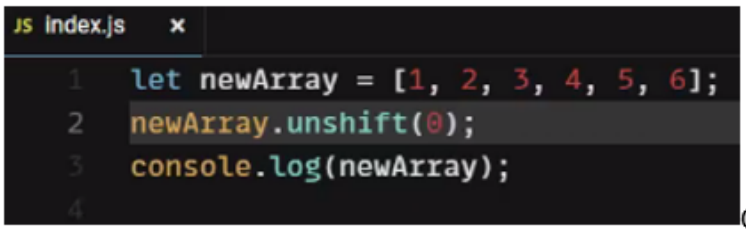
1. pop – It helps us to delete an item from the end of the array.
2. push – This method helps to add an item to the end of an array.
3. shift – This method helps to remove an item from the beginning of an array.
4. unshift – It helps to add an item to the beginning of an array.



Outputs [1, 2, 3, 4, 5]



Outputs [1, 2, 3, 4, 5, 6, 7]



Outputs [0, 1, 2, 3, 4, 5, 6]

5. findIndex – This method is used to return the index.

6. Slice

This method copies the given part of an array and returns the copied part as a new array. It doesn't change the original array. The first parameter **start** is used as the starting point. The second parameter **end** denotes the position and index until the array needs to be sliced.

7. splice – This method is used to modify the contents of an array by removing the existing elements or by adding new elements. It takes 3 parameters. (start position , no of elements to be removed , remaining elements to be added) it changes the original array and returns the removed elements array .

7.a) Removing elements

```
let let sliceArray: number[] 6];
let sliceArray = newArray.splice(1, 2);
console.log(sliceArray);
console.log(newArray);
```

```
▶ (2) [2, 3]
▶ (4) [1, 4, 5, 6]
```

7.b) Adding elements

```
let newArray = [1, 2, 3, 4, 5, 6];
let sliceArray = newArray.splice(1, 2, 7, 8);
console.log(sliceArray);
console.log(newArray);
```

```
▶ (2) [2, 3]
▶ (6) [1, 7, 8, 4, 5, 6]
```

Use .splice(6,0,7,8) to add 7 and 8 at the end of newArray.

8. Split = this method is used for strings . it divides a string into substring and returns them as array.

2 parameters (separator , limit - limits the number of splits(optional)) ;

```
let sentence = "Hello this is UI Class";
let sentSplit = sentence.split(" ", 3);
console.log(sentSplit);
```

```
▶ (3) ["Hello", "this", "is"]
```

9. join() - This method is used to join the elements of an array into a string.

Syntax:
array.join(separator)

```
let sentence = "Hello this is UI Class";
let sentSplit = sentence.split(" ");
let joinArray = sentSplit.join(" ");
console.log(sentSplit);
console.log(joinArray);
```

Important methods

Map

This method is used to create a new array from an existing one, by applying a function to each one of the elements in an array.

```
let arrOne = [1, 2, 3, 4, 5, 6];
let arrTwo = arrOne.map(function (item) {
  return item * 2;
});
console.log(arrTwo);
```

```
let arrOne = [1, 2, 3, 4, 5, 6];
let arrTwo = arrOne.map((item) => item * 2);
console.log(arrTwo);
```

```
▶ (6) [2, 4, 6, 8, 10, 12]
```

Filter

This method takes each element in an array, and it applies a conditional statement against it. If condition is true the elements are pushed to the output array, if false it does not get pushed to output array.

```
let arrOne = [1, 2, 3, 4, 5, 6];
let arrTwo = arrOne.filter((item) => item % 2 === 0)
;
console.log(arrTwo);
```

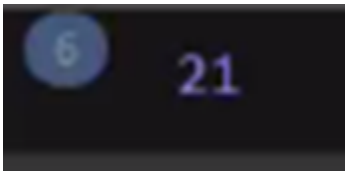
```
▶ (3) [2, 4, 6]
```

Reduce

This method reduces an array down to just one value.

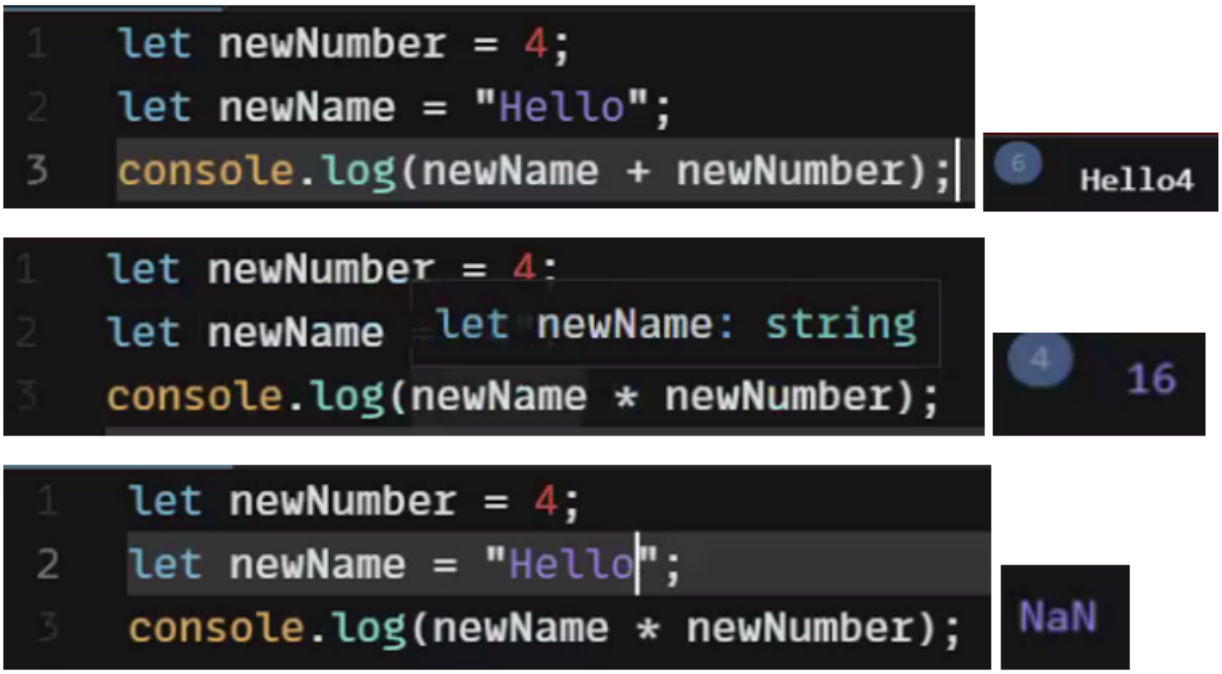
Function === const reducer = (previousValue, currentValue) => previousValue + currentValue;

```
let arrOne = [1, 2, 3, 4, 5, 6];

let arrTwo = arrOne.reduce(function(result,item){
  return result+item;
},0)
console.log(arrTwo);
```



Type Conversion/ Type Coercion

At times, JavaScript automatically converts one datatype to another, and this conversion is known as implicit conversion.



- If a number is added to a string, JavaScript converts the number to a string before concatenation. If one is string, it converts the other to string as well.

Objects

```
const student = {
  firstName: "Tom",
  Age: 30
};
console.log(student.firstName);
```

```
Tom
```

```
const student = {
  firstName: "Tom",
  Age: 30,
  "last Name": "Jerry"
};
console.log(student["last Name"]);
```

```
Jerry
```

```
const student = {
  firstName: "Tom",
  Age: 30,
  "last Name": "Jerry"
};
student.firstName = "Bruto";
console.log(student);
```

```
{firstName: "Bruto", Age: 30, last Name: "Jerry"}
```

```
let student = {
  firstName: "Tom",
  Age: 30,
  "last Name": "Jerry"
};
student.firstName = "Bruto";
student.salary = 10000;
delete student.Age;
console.log(student);
```

```
{firstName: "Bruto", last Name: "Jerry", salary: 10000}
```

for...in loop

It helps us to iterate over the properties of an object

```
let student = {
  firstName: "Tom",
  age: 30,
  "last Name": "Jerry"
};
for (let key in student) {
  console.log(key);
}
```

```
firstName
age
last Name
```

```
let student = {
  firstName: "Tom",
  age: 30,
  "last Name": "Jerry"
};
for (let key in student) {
  console.log(student[key]);
}
```

```
Tom
30
Jerry
```

```
let student = {
  firstName: "Tom",
  age: 30,
  "last Name": "Jerry"
};
let key = "";
for (key in student) {
  console.log(typeof key);
}
```

```
6 string
```

Destructuring:

Array Destructuring : It allows us to destructure properties of an object or elements of an array into individual variables.
-get elements from a given array from beginning .

```
let newArr = [1,2,3,4,5]
let [x,y] = newArr;
console.log(x,y);
```

1 2

```
[x,,y] = newArr; // ignoring the element
console.log(x,y);
```

1 3

Destructuring objects = {}:

```
const students = {
    fname:"tom",
    lName:"jerry",
    age : 20
};

let {fname:firstName , lName:lastName } = students ;
console.log(firstName, lastName);

let {fname , lName } = students ;
console.log(fname, lName);
```

Tom jerry
Tom jerry

const {identifier from obj : alias identifier } = object ;

Default value :

```
let {fname , lName , age  = 30} = students ;
console.log(fname, lName,age);
```

Tom Jerry 20
If age is not passed by obj , then :
Tom jerry 30

Rest parameter :

Rest parameter allows you to represent an indefinite number of arguments as an array
rest(...args)

- Rest parameter should be the last element as shown below .

```
function showRest(a,b,...args){
    console.log(a);
    console.log(b);
    console.log(args);
}

showRest('One','Two', 'Three','Four','Five');
```

```
One
Two
(3) ['Three', 'Four', 'Five']
      0: "Three"
      1: "Four"
      2: "Five"
      length: 3
      [[Prototype]]: Array(0)
```

```
function add(...args){
    return args.reduce( (result,items) => result+items , 0 );
}
console.log(add(1,2,3,4))
```

10

Spread Operator :

it allows you to spread out elements of an iterable object .

For eg. array

Allows you to copy items into another array .

```
let arrone = [1,2,3];
let arrTwo = [4,5,6];
console.log(arrTwo.concat(arrone));

arrTwo = [0 , 0 , ...arrone , 4,5,6];
console.log(arrTwo);
```

[4, 5, 6, 1, 2, 3]

[0, 0, 1, 2, 3, 4, 5, 6]

Asynchronous JS :

setTimeout method = this method executes a block of code after the specified time .

It takes two parameters - (a function containing a block of code to be executed , milli seconds).

```
function greet() {
    console.log("greet");
}

setTimeout( () => {console.log("hello")} , 2000 );
setTimeout(  function(){
    console.log("hi")
} , 3000 )
greet();
```

greet

hello

hi

Promises :

Promise is an object that returns a value we hope to receive in the future .

A promise has 3 states : (can be in any one)

1 .Pending - a promise starts in the pending state . This means the process is not complete .
2. Fulfilled - if the operation is successful . Then the promise ends in the fullFilled state .
3.Rejected - if an error occurs the process ends in the rejected state .

Creating the promise : To create a promise we use the promise constructor
The promise constructor accepts a function as an argument .This function is called the executor.
The executor accepts 2 call by functions i.e resolve & reject

Resolve is called if it is a success and reject is called in case of a failure .

```
let flag = true;

let promExam = new Promise( function (resolve ,reject){
    if(flag){
        resolve("Promise is fullfilled");
    }
    else{
        reject("Promise is failed");
    }
} );


console.log(promExam)
```

Promise chaining -

then method :- this method is used with the callback when the promise is successfully fulfilled or resolved.
onFulfilled() - This callback is called if the promise is fulfilled.
OnRejected() - This callback is called if the promise is rejected.

```
let count = new Promise((resolve, reject) => {
  resolve("Promise is Resolved");
});
count.then((result) => {
  console.log(result);
});
console.log(count);
```

Consuming the Promise

```
let count = new Promise(function (resolve, reject) {
  resolve("Promise Resolved");
});
count.then(function promSuccess(result) {
  console.log(result);
});
console.log(count);
```

The catch method

The catch method is used when the promise is rejected and it schedules a callback for the same.

```
let num = false;
let count = new Promise(function (resolve, reject) {
  if (num) {
    resolve("Fulfilled");
  } else {
    reject("Rejected");
  }
});
count
  .then(function promSuccess(result) {
    console.log(result);
  })
  .catch(function err(result) {
    console.log(result);
  });
console.log(count);
```

Outputs **Rejected**

Outputs **Fulfilled** if num was true.

Finally

This method is used to run a piece of code whether the promise is fulfilled or rejected.

```
let num = true;
let count = new Promise(function (resolve, reject) {
  if (num) {
    resolve("Fulfilled");
  } else {
    reject("Rejected");
  }
});
count
  .then(function promSuccess(result) {
    console.log(result);
  })
  .catch(function err(result) {
    console.log(result);
  })
  .finally(() => {
    console.log("No matter what");
  });
console.log(count);
```

```
▶ Promise {<fulfilled>: 'Fulfilled'}
Fulfilled
No matter what
```

Async Keyword

We use the async keyword with a function in order to convert it into an asynchronous function. The async function returns a promise.

```
async function funName() {
  console.log("This is an Async Function");
  return Promise.resolve("One");
}
funName().then(function (answer) {
  console.log(answer);
});
```

Await

Inside an asynchronous function, the await keyword is used for the asynchronous operation. The await keyword pauses the async function until the promise returns a result.

```
let newPromise = new Promise(function (resolve,
reject) {
  setTimeout(function () {
    resolve("Promise is now resolved");
  }, 4000);
});

async function asynFunc() {
  let result = await newPromise;
  console.log(result);
  console.log("hello");
}
asynFunc();
```

```
let newPromise = new Promise(function (resolve,
reject) {
  setTimeout(function () {
    reject("Promise is not resolved");
  }, 4000);
});
async function anyFunc() {
  try {
    let result = await newPromise;
    console.log(result);
  } catch (error) {
    console.log(error);
  }
}
anyFunc();
```

Fetch API

It provides the fetch() method defined on a window object. This method returns a promise.

Important methods

1. text() - This method resolves the response with a string.
2. json() - This method resolves the promise by parsing the body text as json.

JSON (JavaScript Object Notation)

It is a format for structuring the data.

Characteristics of JSON:

It is human readable and writable.

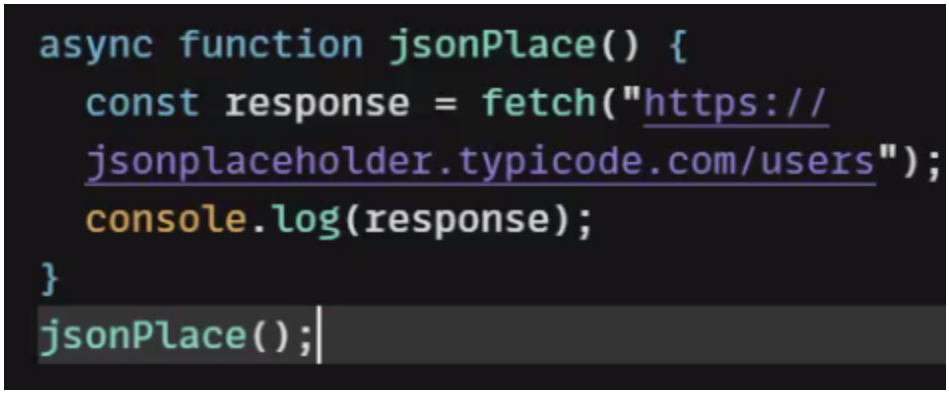
It is lightweight and text based data.

Even though it is derived from JavaScript, it is language dependent.

JSON Syntax Rules

1. Data is in key-value pairs
2. Data is separated by commas
3. Curly braces hold the objects
4. Square brackets hold the arrays.

```
let info = fetch("https://
jsonplaceholder.typicode.com/users")
  .then((response) => response.json())
  .then((result) => console.log(result));
console.log(info);
```



Checkout these in css :

Display - none;

Display: block

Visibility - hidden ;

Position - relative , absolute

Z-index

Transform : rotate(45deg)

Checkout these in JS:

classList.Toggle

TypeScript :

- Superset of Javascript . browser compiles js . but typescript requires a compiler .
- Whatever is written in js can be written in typescript .
- Static typing - defining data type .
- Microsoft developed
- Power of oops

Typescript is nothing but a superset of javascript .It is built on top of javascript . Basically it is JS + static typing .

Why to use typescript :-

1. Typescript adds a type system in order to avoid problems that arise due to dynamic types in JS .
2. It catches the error at compile time .
3. It increases the code quality and readability .
4. TS compiles the code in terms of JS.

Type Annotation :

typeScript uses type annotations to explicitly specify types for identifiers such as variables function , objects etc…

It uses the syntax :type after an identifier as the type annotation . where the type can be any valid type.

Once an identifier is annotated with a type . it can be used as that type only .

Types :

Primitive types :

Numbers: All numbers in TS are either floating point values or big numbers .

Ts also supports hexadecimal literals .

The floating point numbers have the type number , while the big int get the bigint .

String : just like JS , TS also use double/single quotes to enclose a string .

Template Literals :- Template literals are literals delimited with backticks (`), allowing embedded expressions called *substitutions*. ${variable}

```
let num = 1;
let greet = "how are u ";
console.log(`hi i am ${greet} , ${num} = &{33} `);
```

boolean: true/false

array: string[] , number[] etc ..

Club data types in arr : (sting | number) [ ] :

```
let num: string = "one ";
console.log(num);
let newArr : (string | number)[]  = [ "One" , "Two" , "Three" ];
newArr.push(4);
console.log(newArr);
```

Object : the obj type represents everything that isn't a primitive type in ts .For ex . array, tuple etc ..

```
let emp: {
  fname: string;
  lName: string;
  age: number;
} = {
  fname: "Tom",
  lName: "Jerry",
  age: 23
};

console.log(emp);
```

Tuple :

```
let data2: [string, number, number?];
data2 = ["skill ", 20]; // only 2 possible one string one number
console.log(data2);
```

Optional tuple Elements :

Above example ;

Any :

The any type allows us to hold any data.

```
let data: any;
data = ["skill ", 20, true]; // only 2 possible one string one number
console.log(data);
```

Void : the void type denotes absence of having . it is mainly used as the return type of functions

Enum : enum is group of named constant values.

```
enum socialMedia {
    fB,
    insta
  };
  console.log(socialMedia);
```

Conditional :

Functions :

```
function add(a : number , b : number):number {

    return a+b;

}

console.log( typeof(add(5,2)));

let sum:(a:number , b:number) => number ;  // sum is variable , defining
variable type

sum = (a , b ) => a+b ;

console.log(sum(5,8));
```

Optional Parameters (?):

Rest Parameters:

Default Parameters

FORM TAGS :

Used to create an html form for user input .

Components of form :

label : this tag defines a label for different input elements .

for attribute - it indicates the form element that this label describes .

Input : this tag specifies an input field where the user can enter the data .

type attribute :

checkbox type - square box that is ticked when activated . used to allow user to select one or more options from a limited number of choices .

radio type- generally used within radio groups . Only one radio button in the radiogroup can be selected at the same time .

value attribute - the value attribute specifies the value of an input element

name attribute - used to specify a name for an input element .

select tag -> option tag (dropdown)

CSS Topics revised

Centering in CSS

Horizontal:

1. Text-align: center;
2. Margin : 0 auto; and we need to specify the width mandatorily

Vertical:

Position: absolute

Mention top and left as required and use transform

Transform: translatex(50%); // Example

With Flex:

Display: flex

Height: 100vh;

Align-items: center;

Justify-content: center;

Install Node LTS Version

Node.js (nodejs.org)

Run this command in terminal

npm install -g @angular/cli

Create a directory and then run **ng new my-app** in the terminal in the same directory.

Angular

- It is an application framework and development platform for creating efficient and sophisticated single-page apps.
- Angular combines declarative templates, dependency injection, components, services etc.

List of files in a new angular project:

Node_modules: All the dependencies are installed here.

Src: We work in this directory

1. App - It is the root component of our project. A component is a small functionality and it is the building block of our application.

2. Assets - It contains files which can be seen publicly. Ex- font, Images, css, etc.

3. Environments - Contains files for different environments i.e development, productions, etc.
4. index.html - This is the entry point of the html of our application.
5. Main.ts - This is the first ts file to be loaded. It links our components to index.html file.
6. Style.css - Used for global styling.
7. Angular.json - It consists of certain defaults for the angular app. Ex - prefix, favicon, etc.
8. Package-lock.json - Contains information about the dependencies in detail.
9. Package.json - It contains information, app name, version, script dependencies and dev. Dependencies.

ng serve - - open - command to run the app on the local host server

Ng generate component component_name - command to generate a component

Component

How to generate a component?

ng generate component component_name

Shorthand for the above command is - ng g c component_name

A component is a piece of code made to do a specific task and it acts like the building block of our angular application.

Angular Components are a subset of directives, always associated with a template. Unlike other directives, only one component can be instantiated for a given element in a template.

A component is composed of 3 things:

1. HTML Template - It is nothing but the regular html coded with additional angular specific syntax to communicate with the component class. The template defines the layout and the content of the view.
2. Class - A component class is a typescript class that includes properties and methods. It provides data and logic to the view.
3. Meta data - Meta data provides additional information about the component to the angular app. The information includes location of html, css files of the component, selector, etc.

@Component Decorator

A decorator is a function that adds meta data to a class, its methods and its properties.

The components are defined with a @Component class decorator.

It helps angular to treat a class as a component.

Decorator is always prefixed with a '@' and must always be placed before the class definition.

Meta data properties of a component

1. Selector - It specifies the simple css selector. Angular looks for css selector in the template and renders the component there.
2. TemplateUrl - The html template defines our view. It tells angular how to render the component view.
3. StyleUrls - It contains the css styles that the component needs.

Interpolation (String Interpolation)

It is used to display dynamic data on the html template.

It is a one-way data binding technique that is used to transfer the data from the code or class to the html.

The syntax used is as follow {{ }}

.ts file

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'my-app';
  name='James';
}
```

This keyword

Property Binding :

property binding is used to bind html element property to a property in the component

```
<h1>Hello World!</h1>
<h2>{{name}}</h2>
<h3>{{greet()}}</h3>
<button [disabled]="true">Click Me</button>
<input type="text" [readonly]="true" value="Can't Edit">
```

[disabled] = “true” → button

[readonly] = “true” -> input

Event Binding :

```
<button (click) = "subtract()" >-</button>

<h2>{{count}}</h2>

<button  (click) = "add()" > + </button>
```

property Binding vs interpolation :

- property binding is used to to set an element property to a non-string data value .
- property Binding allows us to use boolean values whereas interpolation allows us to use numbers and strings

Style Binding :

```
<h1 [ngStyle] = "{ 'background-color' : 'red' }" >Hello </h1>
```

Class Binding : // css class

```
<h1 [ngClass] = " 'col' ">  Hi </h1>
```

add .col class in component.css file .

conditional classes :

```
<h1 [ngClass] = "classCheck == 'one' ? 'two' : 'three' ">  Hi </h1>
```

ngClass gives us this additional flexibility . So we use [ngClass] instead of only class.

Directives :

directives can help us to extend , change/ modify the behavior of the dom elements .

Component-directive : component is a type of directive with its own template , styles and logic needed for the view .

Attribute Directives : used to modify behavioral appearance .

2 Types :

ngStyle , ngClass .

Structural Directive:

Structural directives are responsible for the html layout i.e they can shape or reshape the html view by simply adding or removing the elements from the dom .

It starts with a * sign

ngFor = similar to the for loop and is used to iterate over a collection of data .

```
<h1 *ngFor = "let item of numbers"  >  the number is {{item}}   </h1>
```

ngIf = it is used to add or remove html elements based on an expression .

```
<h1 *ngIf = " gg==5 ? true:false "  >  hello  </h1>
```

ng-template = it is an angular element which contains the template block .

With <ng-template>, you can define template content that is only being rendered by Angular when you, whether directly or indirectly, specifically instruct it to do so, allowing you to have full control over how and when the content is displayed.

ngIf with **Then Else :**

```
<h1 *ngIf = "show then myThenBlock else myElseBlock">Hello</h1>

<ng-template #myThenBlock>
    <h1>Bye</h1>
</ng-template>

<ng-template #myElseBlock >
    <h1>hi</h1>
</ng-template>
```

ngSwitch = simple implementation of switch case .

```
<div [ngSwitch] = "color"  >
    <h1 *ngSwitchCase="'red'" >Red</h1>
    <h1 *ngSwitchCase="'blue'" >blue</h1>
    <h1 *ngSwitchCase="'yellow'" >yellow</h1>
    <h1 *ngSwitchDefault > Transparent  </h1>
</div>
```

pipes are used for transforming the data .

```
<h1>  {{'MAYANKGUPTA@GMAIL.COM' | lowercase }} </h1>
```

built-in pipes :

lowercase - convert characters to lowercase ;

uppercase -

titlecase-

```
<h1>  {{'laxman bandi' | titlecase }} </h1>
```

currency -

```
<h1>  {{ '123' | currency }} </h1>
```

Date -

```
<h1>{{start | date}}</h1>
```

start = Date.now();

Parameterised Pipes - built on top of built-in types .

we can pass one or more parameters to the built-in pipes .

we use the symbol : to pass the parameters .

currency =

```
<h1> {{ 1234 | currency:'INR' }} </h1>

<h1> {{ 1234 | currency:'EUR' }} </h1>
```

date =

```
<h1> {{ start | date:'long' }}  </h1>

<h1> {{ start | date:'short' }}  </h1>
```

Chaining The pipes :

we can connect multiple types to a particular data input

```
<h1> {{ start | date:'long' | uppercase}}  </h1>
```

Modules :

ngModule - it is a class marked by ngModule decorator

It takes a metadata object that describes how to compile a components template and how to create an injector at runtime .

declarations - it consists of mainly the components ,directives and pipes that belong to this ng module .

imports =

This includes other modules whose exported classes are needed by the component templates - declared in this ngModule.

Providers =

it consists of services which we want to add to - the global collection of services : so that it can become accessible to all parts of the application(components) .

bootstrap = it is the main application view , called the root component .this is the main component of the module which needs to be loaded whenever required .

export = inorder to allow other modules to use components , pipes, directives of the current module , we need to export it first .

Create module :

ng g m <module-name>

g- generate

m-module

create component :

ng g c login/login .

Services :

Services allow us to re-use a piece of code or logic that is used to perform a specific task **.** It can contain both value or function .

how to create service :

ng g s <service-name>

injectable -

it is a decorator used to define a class as a service in angular . It provides the metadata that allows angular to inject it into a component as a dependency .

Dependency injection - It is a technique in which a class receives its dependency from external sources rather than creating them itself .

Routing :

It is a mechanism that is used for navigating b/w the pages and displaying specific components or pages .

Components of the router module :

router - it is an object that enables navigation from one object to another as users perform tasks like clicking on menu links , backslash buttons , etc ..

route - Route tells the angular router which view to display . when a user clicks a link . Every route consists of a path and a component it is mapped to .

The router Object parses and builds the final URL using the route .

Routes - It is an array of route objects .

router Outlet - it is a directive <router-outlet> that serves as a placeholder where the router should display the view .

router-link - It is a directive that binds the html element to a route .

Setting home Page :

Setting a default page :

[routes-redirection]

Setting up the error page :

in app-routing.module.ts

```
const routes: Routes = [

{path:"" , redirectTo: 'main' , pathMatch:"full" } ,
{path:"main" , component: HomeComponent} ,
{path:'home' , component:HomeComponent  }  ,
{path:'about' , component:AboutComponent  } ,
{path:'contact' , component:ContactComponent  } ,
{path:'login' , component:LoginComponent  } ,
{path:'**' , component:TestComponent} ,
  ];
```

in app.component.html

```
<nav class="navbar">
    <a href="" > Logo </a>
    <a href=""  routerLink="home" >Home </a>
    <a href=""  routerLink="about" >About </a>
    <a href=""  routerLink="contact" >Contact </a>
    <a href="" routerLink="login" >login </a>

</nav>

<router-outlet></router-outlet>
```

- Bootstrap , tailwind . can be included in angular .

- **important** tag in css - override all previous styling .

Forms :

Easy error handling with Angular .

2 types .

Template driven form :

They are the simple forms which can be used to develop forms .

advantages of using forms :

- Helps us to initialize the form field and present it to the user .
- Helps us to capture data from the user .
- helps us to track changes made in the fields .
- helps us to validate the inputs .
- helps us to display the errors .

FormsModule : form module provides us with prebuilt services . and it helps us to make all the necessary imports for form implementation . (import FormsModule to app.module.ts)

ngForm - it is the directive which helps to create the control groups inside the form directive .

ngModel - the ngModel directive registers all the input elements with the ngForm .

formControl - it represents a single input field in angular form .

formGroup - it is a collection of formControl .

ngSubmit - we use this event to submit the form data .

```
<form action=""  #login = "ngForm"    (ngSubmit)= "signIn(login)" >

  <input type="text" placeholder="Enter your name"  name="name" ngModel
minlength="7" >
  <br>
  <br>
  <input type="password" placeholder="Enter Password" name="password" ngModel >
  <br>
  <input type="number" placeholder="Enter Age" name="Age" ngModel >
  <br>
  <input type="email" name="email" placeholder="Enter Email" email ngModel >
  <br>
  <button type="submit" [disabled]= "login.valid == false"  >Sign In </button>


</form>
```

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  // title = 'form-app';
  signIn(logIn:any){
    console.log(logIn);
  }
}
```

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { FormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule

  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Validation : Validate certain conditions .

[disabled] = ngForm object.valid == true

classes get added in ngModel as the input gets changed -> ng-touched , ng-invalid , ng-pristine , ng-dirty , ng-untouched

- # is used to give reference to a html element .

displaying the errors . -> using # to give reference

```
<form action=""  #login = "ngForm"  (ngSubmit)= "signIn(login)" >

  <input type="text" placeholder="Enter Your name"  name="name" ngModel
minlength="7" #name = "ngModel" >
  <br>
  <small *ngIf = "name.invalid"  > Enter Valid name   <br>  </small>

  <input type="password" placeholder="Enter Password" name="password"
minlength="8" ngModel #pass = "ngModel" >
  <br>
  <small *ngIf = "pass.invalid"  > Enter Valid password   <br>  </small>

  <input type="text" placeholder="Enter Age" name="Age"  ngModel >
  <br>

  <input type="email" name="email" placeholder="Enter Email" email ngModel
#mail = "ngModel" >
  <br>
  <small *ngIf = "mail.invalid"  > Enter Valid mail   <br>  </small>


  <button type="submit" [disabled]= "login.valid == false  ||
name.value.length == 0 || pass.value.length == 0  "  >Sign In </button>


</form>
```

Two way data binding

Two-way binding gives components in your application a way to share data. Use two-way binding to listen for events and update values simultaneously between parent and child components.

It refers to sharing data between a component class and template.

Syntax - [()] Also called as banana box

```
  <input type="text" placeholder="Enter Your name"  name="name" ngModel
minlength="7" #name = "ngModel" required
  [(ngModel)]= "firstName"  >
  <br>
  <small *ngIf = "name.invalid && name.touched"  > Enter Valid name   <br>
</small>
  <small *ngIf = "firstName"  >  {{firstName }}   <br>  </small>
```

Reactive forms :

These are the forms where we define the structure in the component class.It allows us to create the form controls, form groups and form arrays.

reactiveFormModule :

FormControl:

FormGroup : It represents a collection of form controls. It also keeps track of the value and validation status of the particular group.

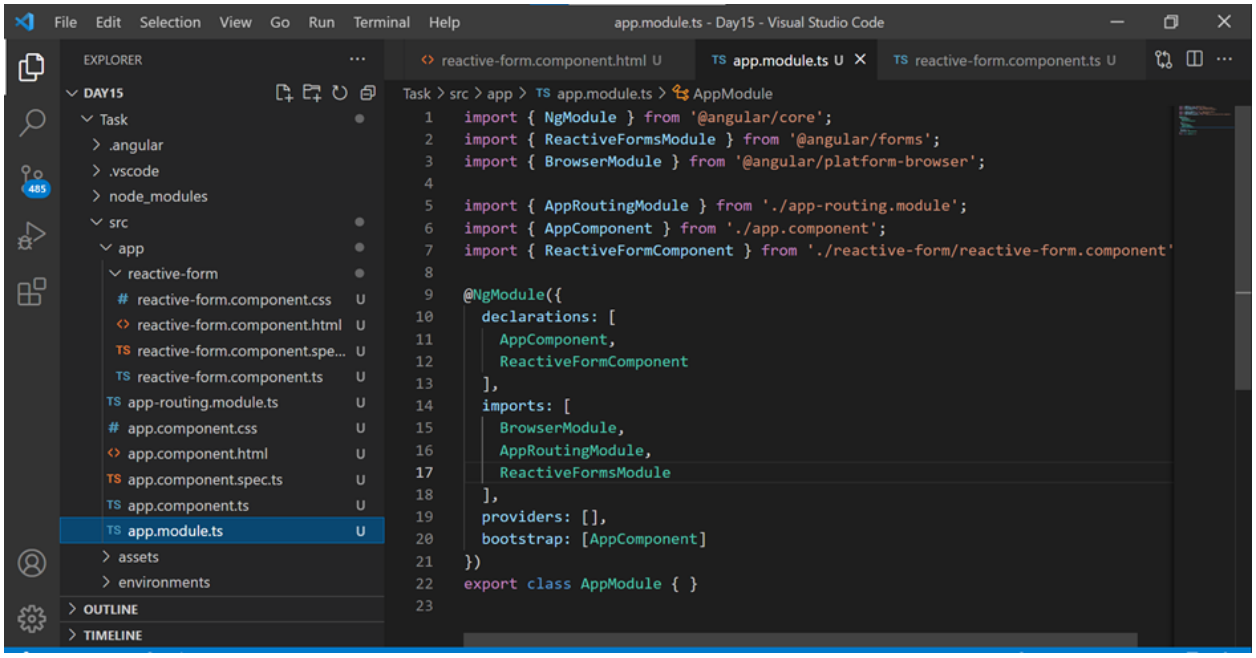
Advantages of Reactive forms:

1. Can handle complex situations.
2. It is scalable and re-usable.
3. It has structured data flow.
4. It is immutable - in terms of data
5. It is flexible - in writing in the component itself .

FormControlName:

It is used to sync a formControl in an existing formGroup to a formControl element by name. Every form field will have this attribute.

Step 1: import reactive form



Step2: Creating form object having form control objects. *.component.js*

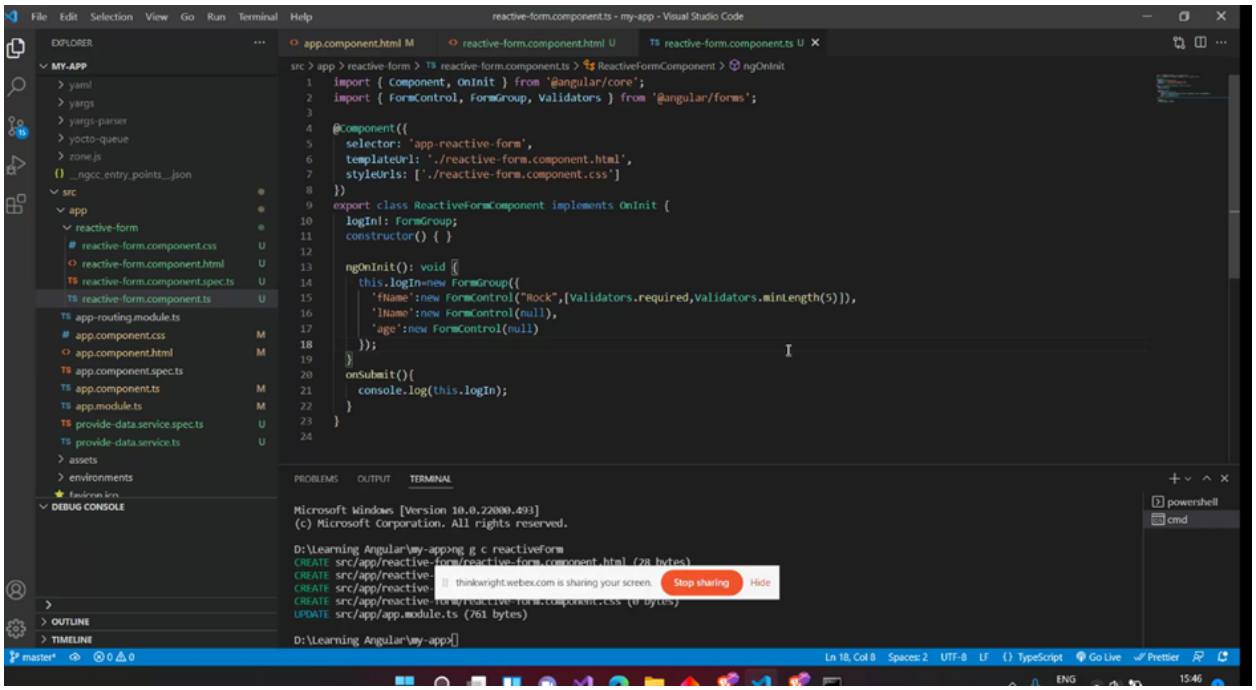
```
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup } from '@angular/forms';

@Component({
  selector: 'app-reactive-form',
  templateUrl: './reactive-form.component.html',
  styleUrls: ['./reactive-form.component.css']
})
export class ReactiveFormComponent implements OnInit {
  login!: FormGroup;
  constructor() { }

  ngOnInit(): void {
    this.login = new FormGroup({
      'fName' : new FormControl(''),
      'lName' : new FormControl(''),
      'age' : new FormControl(''),
```

```
    });

  }

}
```

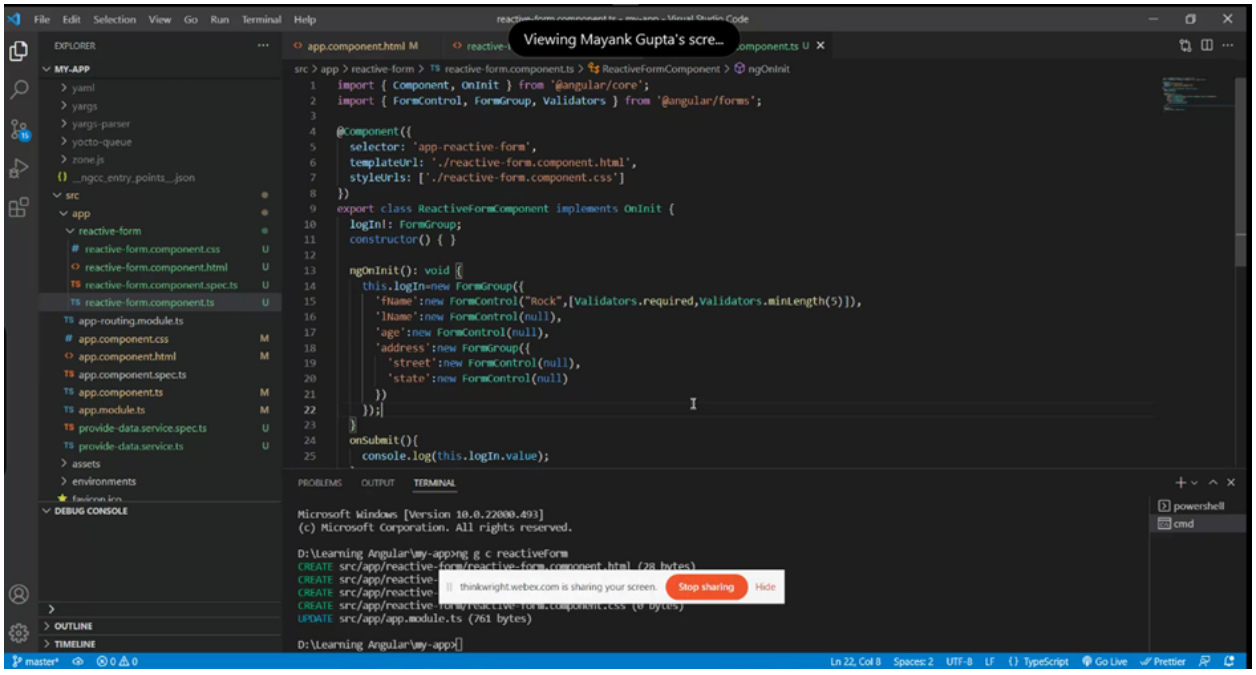
Step 3:

```
<!-- <p>reactive-form works!</p> -->

<form action="" [formGroup] = "myGroup" (ngSubmit)=  "onSubmit()"   >

    <input type="text" formControlName = "fName" placeholder="Enter First
Name">
    <br>

    <input type="text" formControlName = "lName" placeholder="Last Name" >
    <br>
    <input type="text" formControlName = "age" placeholder="Enter  Age" >
    <br>
    <button type = "submit">submit</button>

</form>
```

Form Validation:



FormGroups:



- login.get('name).touched - access to form control

APIs in Angular: for CRUD operations

Rest(Representational State Transfer) API:

It is an API which conforms to the rules of rest architectural structure. An API which is built on this structure is called rest based API.

http Methods:-

Get:
This method is used to read or retrieve representation of a resource.

Post:
This method is used to create new resources.

Put:
This method is used for updating.

Patch:
It is used to modify.

Delete:
Delete the identified data.

API Handling :

HttpModule:
httpModule offers the httpClient and is used to perform http requests and handle responses.

HTTP Client:

It is a built-in service class available in the @angular/common/http package. It will help us to fetch external data, post to it etc.

There are some prerequisites:

Step 1:

import HttpClientModule in the appModule.ts file. IT would be added to the imports.

Step 2:

Create a service and import http client into it.(ng gs serviceName)

service - @injectible - dependency injection

Step 3:

We need to create an instance in the service.ts

Step 4 :

Get the data

```
export class ProvideDataService {

  constructor( private http: HttpClient  ) { }
  getComments(){
       return
this.http.get("https://jsonplaceholder.typicode.com/users").subscribe( (data)
=> console.log(data) );

  }
}
```

Observables : which acts as mediator between client and server

need to subscribe to consume the data .

CRUD-

Create: **Post**: 201 if successful.

Read: **get** :200 if successful. (most important) .

Update: **patch**:204 if successful.

Delete: **delete**:204 if successful.

404 - if wrong URL . Page Not Found .

POST :

HttpClient.post() method performs the **http post** method. This method constructs an observable instance.

HttpClient.post(URL , body , options)

1. **URL:**

   It is the backend service for the end-point of type- string .

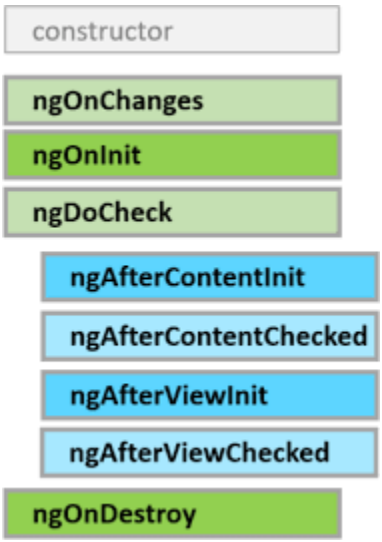
2. **Body:**

   The content to send or replace with of type - any.

3. **Object: (http options of type object)**

The http post returns the observable response of type string

LifeCycle Hooks :



Constructor:

While working with components the first step is to call the constructor. It happens before the implementation of any life-cycle hooks. It is invoked when the class is created. It is used to initialize the class members. It is used for injecting the dependencies in the angular component.

ngOnChanges:

This method is called when any of the properties bound already have their value changed. This method is called every-time whenever there is a change in the value.

@Input:

It is a decorator that marks the property as the input property, i.e it can receive data from the parent component .

ngOnInit:

This is the most usable hook available. This hook is fired only once and it helps us in the initialization of the component and directive.

Constructor vs ngOnint-

Constructor	ngOnInit
IT works before any of the lifecycle hook comes into play.	It fires after constructor and ngOnChanges.
It is used for dependency injection.	It is used for initialization and declaration.
It is only invoked when the component has been initialized.	

ngDoCheck:

This lifecycle hook comes after ngOnInit. It is called to detect and add upon changes that angular can’t or won't detect on its own. It fires with every change detection system.

ngAfterContentInit:

This hook is called after the component's projected content has been fully initialized.

ngAfterContentChecked:

It is called during every changed detection cycle after angular finishes checking of components projected content .

REACT JS :

ReactJS 15-2-22

Angular is a framework and react is a library. React has the concept of virtual DOM. Performance of angular is slow as angular is working on real DOM unlike react which works on virtual DOM. React was created by Facebook. React Native is for mobile App.

Advantages :

- Learning curve is not so steep .
- Fast
- Reusable - component based .
- Tools For debugging - Chrome extension - React developer tool
- Big Supportive community .

Disadvantages :

- Dependency on other tools is limited.
- Rapid changes in implementation .

Why to use react?

- It is a declarative, efficient and flexible JS library.

- It is built with components and components are the key ingredient. Everything is a component in this,no services are there.

Features of React JS:

JSX, It is a JS syntax extension, JSX stands for JavaScript XML. It helps us to write html code in a JS file.

Features

One Way Data Binding: unidirectional flow .

Performance:

JSX makes React faster than normal html and js. Also the virtual DOM gives this the edge over other frameworks or libraries.

Virtual DOM (How React Works):

A virtual DOM is a lightweight virtual JS representation of the DOM. Manipulating DOM is slow but manipulating VDOM is fast as VDOM only changes the section and there is no need to reload an entire page like what a DOM does.

A virtual DOM object has the same property as a real DOM but it lacks the real power to directly change what's on the screen.

When you render a JSX element every single JSX element gets updated. Once the VDOM is updated, react compares the VDOM with a VDOM snapshot that was taken right before the update.

By comparing the new VDOM with the previous version, react figures out exactly which VDOM objects have been changed. This process is known as Diffing.

Once react knows which VDOM has been changed the react only updates that changes on the real DOM.

Installing React LS

write globally- npm install create-react-app -g

Now go to the file/folder where u want the react app and open cmd

Write- create-react-app first-app

Now open VS Code

To start server Write – npm start

INFO: in my-first app .

Robot.txt:

Google spiders crawl through our website and decide what to do with the website. It is used to tell spiders where to crawl or not.

eg . Do Not crawl contact page .

ReportWebVitals - digital marketing stuff

Gitignore:

lists whatever we need to ignore.

YARN:

It is another package manager built by FB. It's faster than npm.

In public we mostly need only index.html.

Class based or State Based Component component:

Install package: vs code extension

ES7+ React/Redux/React-Native snippets

This is initial step

A class component should include extends React.component statement. This statement creates an inheritance to React.Component and gives your components access tp React.Component function.

Render:

This method returns html code.It takes two parameters.First one is that to be rendered and second one is that you want to render.

Syntax: render( ){ }

ReactDOM:

It is a package that provides DOM specific methods .

```
import React , {Component} from 'react';
import ReactDOM from 'react-dom';

class Greet extends React.Component{
  render(){
    return (
      <h1> Hello World  </h1>
    );
  }
}

ReactDOM.render( <Greet name ="lax" /> , document.getElementById('root') );
```

Above was Class based or State Based Component

Above is how you can use render.

JSX(Syntactic sugar):It allows to HTMl kind of code.

Function Based Component / state less components / dumb components:

Advantages:

Syntax is easy, so easy to use. React itself promotes function based components. Class Components are old.

Rfce - shortcut will give a piece of code . / **Rafce**

React components should start with a capital letter .

```
import React from 'react';
import ReactDOM from 'react-dom'

const Element = ()=>{
  //this is a react component that should start with capital letters .
  return (
    <h1>Hello</h1>
  );
}

ReactDOM.render( <Element/> , document.getElementById('root') );
```

Above Element only returns 1 element :

We can't return 2 <h1> but can return 1 div.

React.Fragment:

Allows us to club multiple elements.

```
const Element = ()=>{
  return (
    <React.Fragment>
        <h1>hello</h1>
        <h2>Hi</h2>
    </React.Fragment>

  );
}
```

OR can be written in this way too .

```
 <>
      <h1>hello</h1>
      <h2>Hi</h2>
    </>
```

Rendering dynamic data:

Calling a function / variable :

To put a class in the return part:

Here we have imported bootstrap to change color of first h1

Changing className of return dynamically:

```
const Element = ()=>{
  const fName = "Laxman " ;
  const count = 10 ;
  const nameFunc = () =>{
    return "Mayank"
  }
  return (
    <>
      <h1  className=  { count <5 ? "text-primary" : "text-warning"  } >hello ,
{ count <5 ? "Tom" : "Jerry" }</h1>
      <h2>Hi</h2>
    </>

  );
}
```

TASK:

Print numbers in array:

```
 arr.map( (a)=> <h1>  {a} </h1> )
```

1. Don’t touch index.js
2. Create App.jsx
3. we create a component and inside that create a Navbar.js

Task:

https://www.frontendmentor.io/challenges/base-apparel-coming-soon-page-5d46b47f8db8a7063f9331a0

Handling Events:

Remember when calling a function do not use() or else the function will be fired regardless of if or when it is called.

```
const App = () => {

  const handleClick = () =>{
    console.log("hello");
  }
  return (
  <>
  <button onClick={handleClick}  > Clickme    </button>
  {/* <button onClick={handleClick()}  > Clickme    </button>   // doesnt work
*/}
```

```
  </>

  );
}

export default App
```

To use images in public folder

16-2-22

State:

State is a built-in react object that is used to contain data or information about the component.

useState:

The useState hook in react allows us to track state in a function component.

Syntax:

```
const [count , setCount] = useState(0);
```

The first element is the initial state. The second element is a function that is used for updating the state.

Final code: Count + 1

```
import React , {useState} from 'react'

const Count = () => {
```

```
    const [count , setCount] = useState(2);
    const handleClick = () => {
      setCount(count+1) ;
      // count+=1;
      console.log(count);
    }

    return (
      <>
        <h1> {count} </h1>
        <button onClick={handleClick} > Click Me  </button>
      </>
    )
  }

export default Count
```

When count is object :

```
import React , { useState} from 'react'

const Count2 = () => {

    const [count , setCount] = useState({
        count1:0,
        count2:1
    });

    console.log(count);
    const handleClick = () => {
      setCount( { ...count ,  count1: count.count1 +1 } ) ;
      // count+=1;
      console.log(count);
    }

    const handleClickTwo = () => {
        setCount( { ...count, count2:count.count2 +1 } ) ;
        // count+=1;
        console.log(count);
      }

    return (
```

```
      <>
        <h1> {count.count1} </h1>
        <button onClick={handleClick} > Click Me  </button>
        <h2>  {count.count2} </h2>
        <button onClick={handleClickTwo} > Click MeTwo </button>
      </>
    )
  }

export default Count2
```

Counter:

```
import React from 'react'
import { useState } from 'react'

const Counter = () => {

 const [count , setCount] = useState(0);

 const Increment = ()=>{
     setCount(count+1);
 }
 const Decrement = ()=>{
    setCount(count-1);
}
  return (
    <>
        <button onClick={Increment} > + </button>
        <h2>{count}</h2>
        <button onClick={Decrement} > - </button>


    </>
  )
}
```

```
export default Counter
```

Passing arguments to a function with events:

```
import React from 'react'
import { useState } from 'react'

const Counter = () => {

 const [count , setCount] = useState(0);

 const Increment = ()=>{
     setCount(count+1);
 }
 const Decrement = ()=>{
    setCount(count-1);
}


const handleClick = ( val ) =>{
    if(val == 1){
        setCount(count+1);
        console.log(count);
    }
    else{
        setCount(count-1)
        console.log(count);
    }
```

```
}
  return (
    <>
        <button onClick={ () => handleClick(1)} > + </button>
        <h2>{count}</h2>
        <button onClick={  () => handleClick(2)  } > - </button>


    </>
  )
}

export default Counter
```

Props: (props stand for properties)

Props are the arguments passed into react components. Props are passed to components via html Attributes.

It is an object that stores the value of attributes of a tag. It allows us to pass data from one component to another component.

- it's nothing but an object .

Parent:

Child:

Twitter Clone :

You won't be able to send details in the above.

To avoid DRY and less code props can be used.

Properties passed from the parent to child within the components from one to other.

```
import React from 'react'
import Twitter from './Twitter'

const Tweets = () => {

    const tweet = [
        {
            id:1,
            name: "Elon Musk" ,
            handle : "elonmusk" ,
            likes : 10,
            tweet : " lorem " ,
        },
        {
            id:2,
            name: "Laxman BAndi " ,
            handle : "laxman.ly" ,
            likes : 20,
            tweet : " lorem " ,
        },
        {
            id:3,
            name: " Mayank Gupta " ,
            handle : "maya" ,
            likes : 40,
            tweet : " I train studs  " ,
        }
    ]


    const renderTweets = () => {
        return tweet.map( item => {
```

```
            return <Twitter objects  = {item}  />
        }  )
    }
  return (
    // tweet.map( x => <Twitter  name = {x.name} handle = {x.handle} tweet =
{x.tweet}   /> )

    <>
       {renderTweets()}
       <h1></h1>
    </>
  )
}

export default Tweets
```

17-2-22

Setting Parent state from children Component in state .

Below Code we can access parent function through child

Altering parent data/state from child:

API handling in React: (Asynchronous by nature)

useEffect():

To handle side effects of class, components etc.

It is a hook that allows you to perform side effects in your components.

UseEffect() is a function which takes two arguments.The first argument passed to useEffect is a function called effect and the second argument (which is optional) is an array of dependencies.

#imp: Controlled and Uncontrolled input

The html form elements such as <input><textarea>& <select> typically maintain their own state and update it based on user input.

To prevent reload on submit: **preventDefault( )**

Input box value onChange/ Handling state of an input box:

Use formic or react form for form in react:

21-2-22

Context API: https://tdsjis.csb.app/

Solution for prop drilling:

Context provides a way to pass data through the component tree without having to pass props down manually at every level.

createContext:

It creates a context object when react renders a component that subscribes to this context object. It will read the current context value from the closest matching provider above it in the tree.

Provider:

Every context object comes with a provider react component that allows consuming components which subscribe to context changes.

The provider component accepts a value prop to be passed to consuming components that are descendants of this provider.

One provider can be connected to multiple consumers. Providers can be nested to override values deeper within the tree.

useContext:

In order to use the context in a child component we need to access it using the useContext hook.

This hook accepts a context object (The value returned from React.createContext) and returns the current context value for that context.

The current context value is determined by the value prop of the nearest <myContext.Provider> above the calling component in the tree.

https://sxneek.csb.app/

index.js

```
import { StrictMode } from "react";

import ReactDOM from "react-dom";



import App from "./App";



const rootElement = document.getElementById("root");

ReactDOM.render(

  <StrictMode>

        <App />

  </StrictMode>,

  rootElement

);
```

App.jsx

```
import AddEntry from "./components/AddEntry";
```

```
import Entries from "./components/Entries";
import Summary from "./components/Summary";
import TotalBalance from "./components/TotalBalance";
import { ExpenseProvider } from "./expense-context";
import "./styles.css";

export default function App() {
  return (
        <ExpenseProvider>
        <div className="container justify-content-center">
        <TotalBalance />
        <Summary />
        <AddEntry />
        <Entries />
        </div>
        </ExpenseProvider>
  );
}
```

Expense-context.js

```
import { createContext, useState } from "react";


export const ExpenseContext = createContext();


export const ExpenseProvider = (props) => {
  const [entries, setEntries] = useState([
        { id: 1, description: "Pizza", amount: -99 },
        { id: 2, description: "Samosa", amount: -7 },
        { id: 3, description: "Dosa", amount: -15 },
        { id: 4, description: "Salary", amount: 200 },
        { id: 5, description: "Stocks", amount: 50 }
  ]);
  const handleDelete = (id) => {
        setEntries(entries.filter((entry) => entry.id !== id));
  };
  return (
        <ExpenseContext.Provider value={{ entries, handleDelete }}>
```

```
        {props.children}
        </ExpenseContext.Provider>
  );
};
```

TotalBalance.js

```
import { useContext } from "react";
import { ExpenseContext } from "../expense-context";

const TotalBalance = () => {
  const { entries } = useContext(ExpenseContext);
  console.log(entries);
  const tBalance = entries.reduce(
        (previousValue, currentValue) => previousValue + currentValue.amount,
        0
  );
  return (
        <div className="">
```

```
        <h2 className="display-3 text-center">${tBalance}</h2>
        </div>
  );
};
export default TotalBalance;
```

Summary.js

```
import { useContext } from "react";
import { ExpenseContext } from "../expense-context";

const Summary = () => {
  const { entries } = useContext(ExpenseContext);
  let income = 0;
  let expense = 0;
  entries.forEach((element) => {
        if (element.amount > 0) {
        income += element.amount;
        } else {
```

```
        expense += element.amount;
        }
});
console.log(entries);
return (
        <div>
        <div className="row mt-3">
        <div className="col-sm-6">
        <div className="card border-success">
        <div className="card-body text-center ">
        <h5 className="card-title text-center">Income</h5>
        <h4 className="text-success-display-5">${income}</h4>
        </div>
        </div>
        </div>
        <div className="col-sm-6 mt-2">
        <div className="card border-danger">
        <div className="card-body text-center">
        <h5 className="card-title">Expense</h5>
```

```
        <h4>-${expense * -1}</h4>
        </div>
        </div>
        </div>
        </div>
        </div>
  );
};
export default Summary;
```

AddEntry.js

```
const AddEntry = () => {
  return (
        <div className="border border-info rounded p-4 mt-3">
        <form action="" className="form-group">
        <div className="form-group">
        <label htmlFor="description">Description</label>
        <input type="text" id="description" className="form-control" />
```

```
        </div>
        <div className="form-group">
        <label htmlFor="description">Amount</label>
        <input type="text" id="Amount" className="form-control" />
        </div>
        </form>
        <button className="btn btn-primary mt-3">Add Entry</button>
        </div>
  );
};
export default AddEntry;
```

Entries.js

```
import { useContext } from "react";
import { ExpenseContext } from "../expense-context";
import Entry from "./Entry";

const Entries = () => {
```

```
  const { entries } = useContext(ExpenseContext);

  return (
        <div className="mt-3">
        <h4 className="display-6">Entries</h4>
        <hr />
        <div className="list-group">
        {entries.map((el) => {
        return <Entry entry={el} />;
        })}
        </div>
        </div>
  );
};

export default Entries;
```

Entry.js

```
import { useContext } from "react";

import { ExpenseContext } from "../expense-context";
```

```
const Entry = (props) => {
  const { handleDelete } = useContext(ExpenseContext);
  return (
        <div>
        <li className="list-group-item list-group-item-action">
        {props.entry.description}
        <button
        className="btn-close float-end"
        onClick={() => {
        handleDelete(props.entry.id);
        }}
        ></button>
        <span className="float-end">${props.entry.amount}</span>
        </li>
        </div>
  );
};
export default Entry;
```

22-2-22

useReducer: https://69liw4.csb.app/

Banking.jsx:

```
import { useContext, useState } from "react";
import { BankingContext } from "./banking-context";


const Banking = () => {
  const { balance, dispatch } = useContext(BankingContext);
  const [amount, setAmount] = useState("");
  return (
        <div className="">
```

```
<h1>Balance is {balance}</h1>

<input

type="text"

value={amount}

onChange={(e) => {

setAmount(e.target.value);

}}

/>

<button onClick={() => dispatch({ type: "WITHDRAW", amount })}>

Withdraw

</button>

<button

onClick={() => dispatch({ type: "DEPOSIT", amount }, setAmount(""))}

>

Deposit

</button>

<button>Fixed Deposit</button>

<button>Delete Account</button>

</div>
```

```
  );

};

export default Banking;
```

Redux:

It is a state management library that helps you to manage state in your application. It is not specific to react, can be used with other frameworks or libraries like Angular, vue, vanilla js,flutter etc.

Advantages of redux:

1. It makes the state predictable- You define how to extract the values your component needs from redux, and your component updates automatically as needed. State is immutable.
2. Maintainability
3. It is centralized.
4. It allows time traveling.
5. The redux dev tools make it easy to debug the application(In terms of state).
6. It is flexible- It works with any UI layer, it separates the UI from the logic.

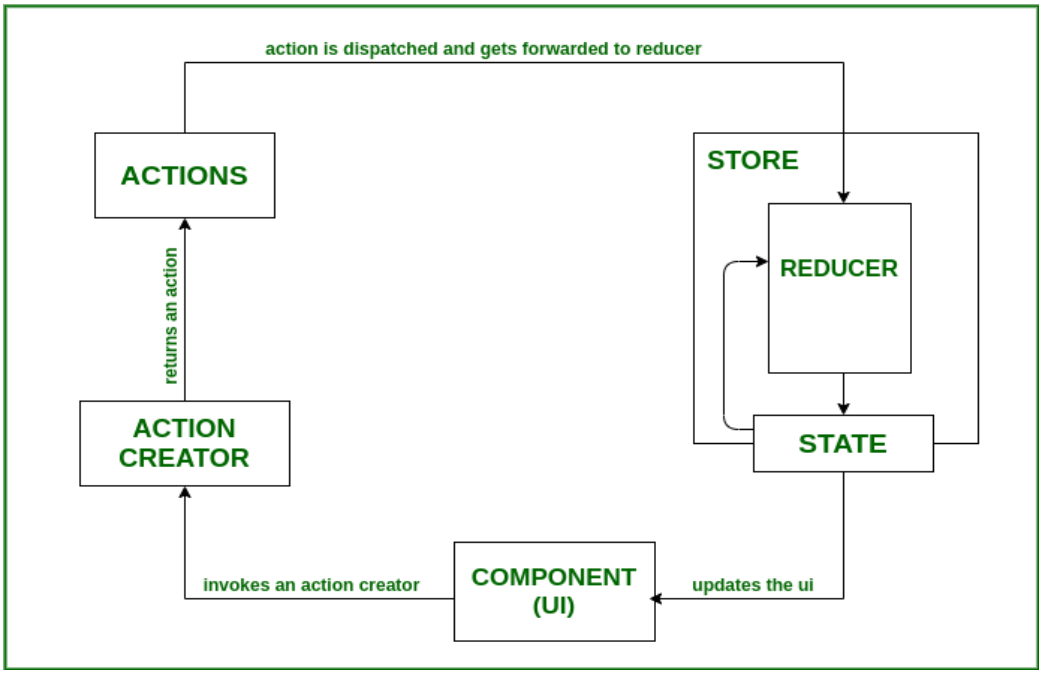


Fig: Typical flow of data in react with redux app

Core principle of redux:

The state of your whole application is stored in an object tree within a single store that acts as a single source of truth for your application.

To specify how the state tree is transformed by actions you write pure reducer functions. The state is read only and the required changes can be made by descriptive action.

Action:

It is an obj that contains a property called type which signifies the type of action that needs to be performed, an optional payload which provides the extra information about the action.

It carries a payload information from your application to your store.

It tells you about the changes or kind of action but it doesn't really make the changes.

Action types:

It is a type of string constant.

Action Creators:

It is a function that creates an action object.

```
const changeWheel = (value) => {
  return {
        type: 'CHANGE_WHEEL',
        value
  }
}
dispatch(changeWheel())
```

Reducer:

It is a pure function which specifies how the application state changes in response to an action.

It takes 2 parameters that are action to be performed and the action to be performed and returns the new state. It handles the action dispatched by the components.

The reducer does not manipulate the original state passed to them, but it makes their own copies and updates them.

Example:

```
// Use the initialState as a default value
export default function appReducer(state = initialState, action) {
  // The reducer normally looks at the action type field to decide what happens
  switch (action.type) {
    // Do something here based on the different types of actions
    default:
      // If this reducer doesn't recognize the action type, or doesn't
      // care about this specific action, return the existing state unchanged
```

```
    return state
  }
}
```

Store:

A store is a state container which holds the application's state. It is an obj that brings all components to work together

Example: import { createStore } from 'redux'

```
import rootReducer from './reducer'

const store = createStore(rootReducer)

export default store
```

Calorie tracker using redux: https://1gpdd1.csb.app/

Source code of Calorie Tracker:

in action folder

Action.js >>

```
export const addEntry = (entry) => {
  return {
    type: "ADD_ENTRY",
    entry: entry
  };
};

export const deleteEntry = (id) => {
```

```
  return {
    type: "DELETE_ENTRY",
    id: id
  };
};
```

---

in components folder

in AddEntry.jsx >>

```
import { useState } from "react";
import { useDispatch } from "react-redux";
import { addEntry } from "../actions/Actions";

const AddEntry = () => {
  const [item, setItem] = useState("");
  const [calories, setCalories] = useState(0);
  const dispatch = useDispatch();
  const handleSubmit = (e) => {
    e.preventDefault();
    dispatch(addEntry({ item, calories }));
    setItem("");
    setCalories(0);
```

```
};
return (
  <div className="border border-info rounded p-4 mt-3">
    <form action="" className="form-group" onSubmit={handleSubmit}>
      <div className="form-group">
        <label htmlFor="description">Food Item</label>
        <input
          type="text"
          id="description"
          className="form-control"
          value={item}
          onChange={(e) => setItem(e.target.value)}
        />
      </div>
      <div className="form-group mt-2">
        <label htmlFor="description">Calories</label>
        <input
          type="text"
          id="Amount"
          className="form-control"
          value={calories}
          onChange={(e) => setCalories(e.target.value)}
        />
      </div>
      <button className="btn btn-primary mt-3">Add Entry</button>
```

```
      </form>
    </div>
  );
};
export default AddEntry;
```

.....

in Entries.jsx >>

```
import { useSelector } from "react-redux";
import Entry from "./Entry";

const Entries = () => {
  const entries = useSelector((state) => state.calorie);
  console.log(entries);
  return (
    <div className="mt-3">
      <h4 className="display-6">Entries</h4>
      <hr />
      <div className="list-group">
        {entries.map((el) => (
          <Entry entry={el} />
        ))}
      </div>
    </div>
```

```
  );
};
export default Entries;
```

.....

in Entry.jsx >>

```
import { useDispatch } from "react-redux";
import { deleteEntry } from "../actions/Actions";

const Entry = (props) => {
  const dispatch = useDispatch();
  const handleDelete = () => {
    dispatch(deleteEntry(props.entry.id));
  };
  return (
    <div className="list-group-item list-group-item-action d-flex justify-content-between mt-2">
      <div className="">{props.entry.item}</div>
      <div className="">{props.entry.calories} kcal</div>
      <button className="btn-close" onClick={handleDelete}></button>
    </div>
  );
};
export default Entry;
```

.....

in TotalCalories.jsx >>

```
const TotalCalories = () => {
  return <div className="display-5 text-center">Total Calories</div>;
};
export default TotalCalories;
```

---

in reducers folder

in Reducer.js >>

```
import { combineReducers } from "redux";

const initState = [];
export const calorieReducer = (state = initState, action) => {
  switch (action.type) {
    case "ADD_ENTRY":
      return [
        ...state,
        { ...action.entry, id: Math.floor(Math.random() * 99) }
      ];
    case "DELETE_ENTRY":
      return state.filter((item) => item.id !== action.id);
```

```
    default:
      return state;
  }
};
export const rootReducer = combineReducers({
  calorie: calorieReducer
});
```

---

in App.js >>

```
import AddEntry from "./components/AddEntry";
import Entries from "./components/Entries";
import TotalCalories from "./components/TotalCalories";
import "./styles.css";

export default function App() {
  return (
    <div className="container">
      <TotalCalories />
      <AddEntry />
      <Entries />
    </div>
  );
}
```

---

in index.js >>

```
import ReactDOM from "react-dom";
import { Provider } from "react-redux";
import { createStore } from "redux";

import App from "./App";
import { rootReducer } from "./reducers/Reducer";

const store = createStore(rootReducer);

const rootElement = document.getElementById("root");
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  rootElement
);
```

Link for Redux: **https://1gpdd1.csb.app/**

Context API: **https://sxneek.csb.app/**

Reducer + Context API: **https://69liw4.csb.app/**

`Headers`

Represents response/request headers, allowing you to query them and take different actions depending on the results.

`Request`

Represents a resource request.

`Response`

Represents the response to a request.