Trailblazer COMMUNITY GROUPS

salesforce

Preparing JavaScript Certification for LWC Development

20th March 2021
7:30PM IST

Salesforce Developer Group, Kolkata, India

Our Speakers

Gaurav Kheterpal

MTX Group

CTO

Avijit Chakraborty

Solution Architect

Santanu Pal

Cognizant

Technical Architect

Santanu Boral

Tavant

Sr. Technical Architect

Agenda

- Why JavaScript Certification?
- Exam Structure
- Listen to experts who have passed the JavaScript Certification
- How this preparation benefits LWC development (solving real life Use Cases using ECMAScript, when to do what)

By the end of this session, you will overcome business challenges applying the knowledge of JavaScript.

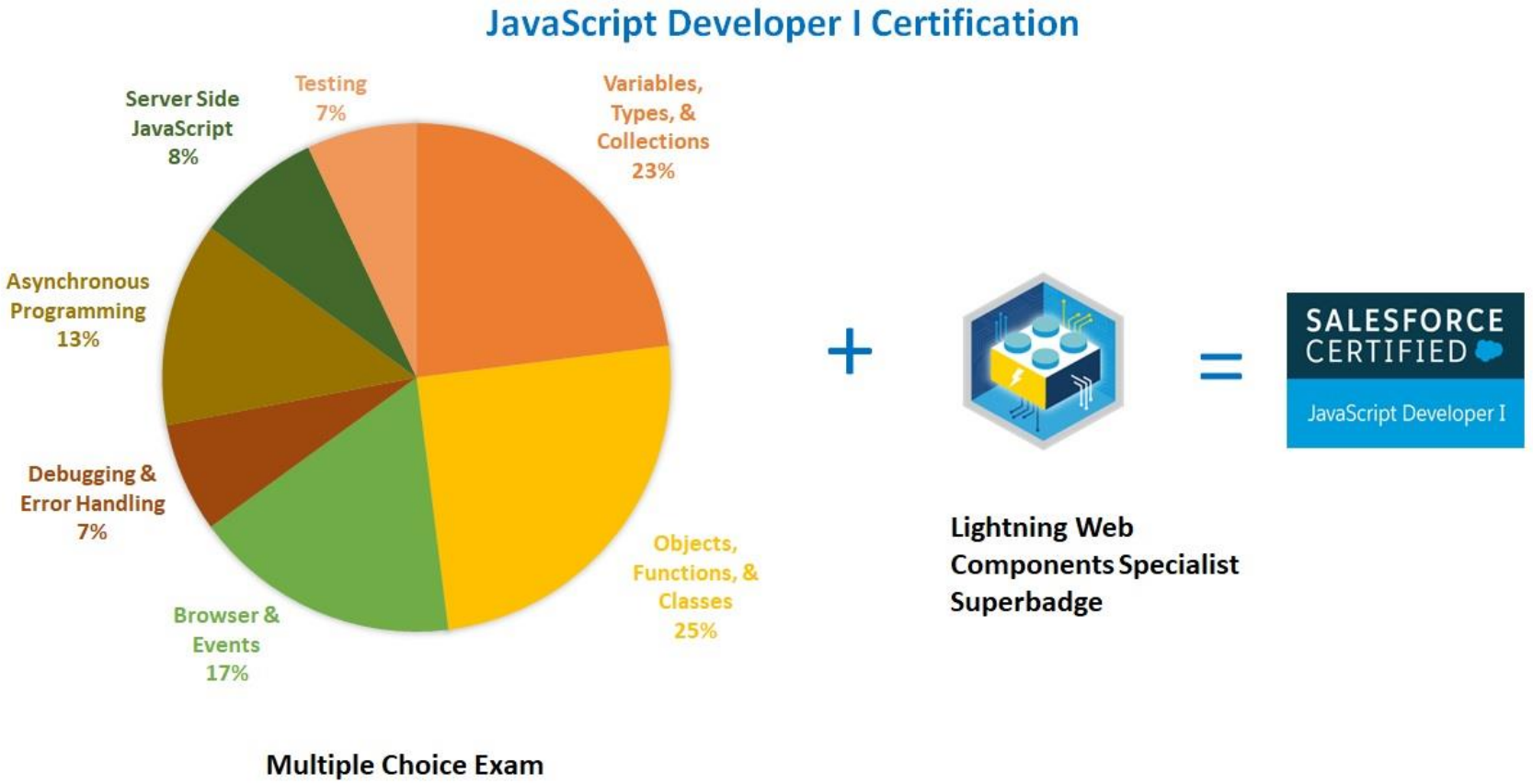
Surprise

Give away Free Voucher ($200)

Why JavaScript Developer 1 Certification?

- With the introduction of Lightning Web Components where JavaScript is a prime language to develop components.
- There are no other Salesforce provided certifications on JavaScript till date
- This is entry level certification for candidates who wants to develop front-end and/back-end JS App for web stack.
- During preparation, developers will learn JavaScript as well as LWC
- As a developer or team lead, this knowledge will help to accelerate the project deliveries and maintenance in an efficient way.
- It has a huge demand in the market and employers are looking this as key certification

Exam Structure



MCQ Exam Details

- 65% passing score
- 105 minutes duration
- 65 questions (5 questions unscored)
- No pre-requisites

Link to Superbadge

How JavaScript fits into LWC

- Lightning Web Components are the lightweight frameworks built on web standards. It also supports standard Web Component framework.
- Lightning Web Components are custom HTML elements built using HTML and modern JavaScript libraries (including latest ESXX features)
- Lightning Web Components are UI framework to develop Desktop and Mobile applications.
- Lightning Web Components is open source, so enterprise-ready web components on any platform, not just Salesforce
- LWC components comprise of HTML, JS and meta file

Let’s first listen to experts

- Motivation to take this exam
- Timeframe to prepare this exam
- Any Challenges faced during preparation or during exam
- Useful tips to prepare
- How it helps them on LWC Development

The Mental JavaScript Console


```
> typeof NaN
< "number"
> 9999999999999999
< 10000000000000000
> 0.5+0.1==0.6
< true
> 0.1+0.2==0.3
< false
> Math.max()
< -Infinity
> Math.min()
< Infinity
> []+[]
< ""
> []+{}
< "[object Object]"
> {}+[]
< 0
> true+true+true===3
< true
> true-true
< 0
> true==1
< true
> true===1
< false
> (!+[]+[]+![]).length
< 9
> 9+"1"
< "91"
> 91-"1"
< 90
> []==0
< true
```

Practice, Practice, Practice!

```
29
30      (function(){
31        console.log((![] + [])[+[]] +
32        (![] + [])[+!+[]] +
33        ([![]] + [][[]])[+!+[] + [+[]]] +
34        (![] + [])[!+[] + !+[]])
35        // JS, wut?
36      })();
37
```

Variables, Types and Collections (22%)

Variable Declarations

var, let and const

Declaration	Usage	Initialization	Variable Hoisting (Use before declare)	Scope

var	var x; //undefined
	var x = 10;
		Optional	console.log(x); //undefined
			var x = 10;
				Function

let	let str = 'tom';
	let name; //undefined
		Optional	console.log(x); //ReferenceError
			let x = 10;
				Block

const	const x = 10;
	x = 7; //can't reassign
		Mandatory	console.log(x); //ReferenceError
			const x = 10;
				Block

No declaration	x = 7; //same as below
	console.log(window.x);
		Optional	console.log(x); //ReferenceError
			x = 10;
				Window or global

Types

Primitive Data Types – Boolean, number, string, symbol, null, undefined

Primitive Wrapper Objects – Object wrapper can contain properties

Type Casting – explicit conversion from one to another

typeof – returns a string indicating the type of value

instanceof – checks the value is instance of an object

Type Coersion – implicit conversion of values

Equality Operators –

- Abstract Comparison ==
- Strict Comparison === (compare types & values of primitive)

falsy – false, 0, NaN, undefined, null, (''), (“”)

```
Boolean b = new Boolean('false');

Number num = new Number(9.08);
```

```
typeof ('99') // "string"

const q = new Number ('10');
q instanceof Number; //true
```

```
10 + '2' + undefined; // "102undefined"
true + 10 * 2; //21
'bat' + null; // "batnull"
"35" – 5; //30
```

```
if (0){
   //this block will never execute
}
```

Using String

String

- Single and double quotes, both are valid
- Template literals are allowed within back tick

```
const city = ‘Kolkata’;
console.log (`Our city is ${city}`);
```

String Methods

- **indexOf** – index of first occurrence of substring
- **substring** - returns a substring
- **includes** – returns a boolean if string is present
- **trim** – removes leading & trailing spaces
- **slice** – extracts a section of string & returns a new string

```
const sentence = 'The quick brown fox jumps over the lazy dog.';
const word = 'fox’;
console.log (sentence.indexOf(word)); //16

console.log(sentence.substring(1,3)); //he

console.log(sentence.slice(4,19)); // "quick brown fox"
```

Play with Arrays

Array - Stores multiple values into single variable

```
let fruits = ['Apple', 'Banana', 'Orange']; //single dimensional array
let fruits = new Array ('Apple', 'Banana', 'Orange');
let arr = [ ['a', 1], ['b', 2], ['c', 3]]; //multi-dimensional array
```

```
//following creates array taking each character
let fruits = Array.from ('Apple'); // ["A","p","p","l","e"],
let arr = Array.of(5); //[5], here 5 is value of 0th index
let arr2 = Array (3); //[undefined, undefined, undefined] , creates array with size 3
Array.isArray(fruits); //true
```

Looping through an array

for…in (index wise)

```
let fruits = ['Apple', 'Banana', 'Orange'];
for (let x in fruits) {
    console.log(fruits[x]);
}
// Apple, Banana, Orange
```

for…of (element wise)

```
let fruits = ['Apple', 'Banana', 'Orange'];
for (let x of fruits) {
    console.log(x);
}
//Apple, Banana, Orange
```

Traditional for loop

```
const arr = [1, 4, 9, 16];
for (let i=0; i< arr.length; i++){
                console.log(arr[i]);
}
//1,4,9,16
```

for…each (operates on function)

```
[2, 5, 9].forEach(logArrayElements);
function logArrayElements(element, index, array) {
  console.log('a[' + index + '] = ' + element);
}
//a[0] = 2, a[1] = 5, a[2] = 9
```

Arrays with Real Life Use Case *Creating and returning new Array (original array content does not change)*

Array.map – creating an upon looping through the elements

```
@wire(getAllProducts)
wiredAllProducts({ error, data }) {
        if (data) {
                data.map(element=>{
                        this.listItems = [...this.listItems,{value:element.Id,
                                                label:element.Name}];
                });
        }
}
```

```
const arr = [1, 4, 9, 16];

// pass a function to map
const mapA = arr.map(x => x * 2);

console.log(mapA);
// expected output: Array [2, 8, 18, 32]
```

Array.filter – if a datatable row to be removed based on recordId

```
handleRemoveComboItems(event){
        const removeItem = event.target.dataset.item; //"0032v00002x7UEHAA2"
        this.listValues = this.listValues.filter(item => item != removeItem);
}
```

```
const arr = [1, 4, 9, 16];

// pass a function to map
const mapA = arr.filter(x => x % 2);

console.log(mapA);
// expected output: Array[4,16]
```

Array.slice – datatable with pagination

```
this.data = this.items.slice(this.startingRecord, this.endingRecord);
```

For example, on 2nd page, label will be shown as => "**Displaying 6 to 10 of 23 records. Page 2 of 5**"

page = 2; pageSize = 5; startingRecord = 5, endingRecord = 10

so, slice(5,10) will give 5th to 9th records.

```
const arr = [1, 4, 9, 16];

console.log(arr.slice(1,3)); //final index omitted

// expected output: Array[4,9]
```

Array Methods where it changes original Array

sort – returns sorted array	**splice** – changes the content by adding or removing elements	**reduce –** executes reducer function on each element resulting single output value.	**push –** add elements(s) at end.

const arr = [1, 4, 9, 16];
console.log(arr.sort());
//Array[1,16,4,9]
	const arr = [1, 4, 9, 16];
	//replaces first element with 5
	arr.splice(0,1,5);
	console.log(arr); //Array[5,4,9,16]
		const arr = [1, 4, 9, 16];
		const reducer = (acc, curr) => acc + curr;
		// 1 + 4 + 9 + 16
		console.log(arr.reduce(reducer)); //output: 30
			const arr = [1, 4, 9, 16];
			arr.push(25); //Array[1,4,9,16,25]
			arr.pop(); //removes last element
			refer shift, unshift functions

Collections – Map & Set

MAP – holds key/value pair.	**SET –** holds unique values (no duplicates)

let vmap = new Map ([['a', 1], ['b', 2], ['c', 3]]);
vmap.set('b',10); //assigns values based on key
vmap.get('c'); //get the values based on key
vmap.has('a'); //check existence
Refer: add, delete, keys, values, forEach functions on Map
	let pSet = new Set([1,4,9,4,16]);
	console.log(Array.from(pSet.values())); //Array[1,4,9,16]
	pSet.has(16); //check existence
	pSet.size(); //size of array, output 4
	Refer: add, delete, keys, values, forEach functions on Set

JSON

Serializing objects, arrays, numbers, strings, booleans and nulls

JSON.parse – parse a JSON string and converts JavaScript value or object.	**JSON.stringify** – converts JavaScript object or value to JSON String.

const json = '{"result":true, "count":42}';
const obj = JSON.parse(json);
console.log(obj.result); //true
	console.log(JSON.stringify([new Number(3), new String('false'), new Boolean(false)]));
	//expected output: "[3,"false", false]"

- If we want to send a collection to Apex Class from JS Controller. we have to transform to String by using JSON.Stringify() and send to Apex Class.
- It is a common way of exchanging request and response for Webservice Integration.

Objects, Functions and Classes (25%)

Objects

- **Property** is an association between name (or key) and value.
- **Objects** are nothing but the collection of properties.

Features

- Objects are mutable
- Object is a non-primitive data type in JavaScript
- Objects can be inherited from another Object
- Gets their own variable context when created
- Every object has __proto__ Object property which refers Parent object.
- Objects are passed by reference; primitives are passed by value

Object

```
let emp = {
    name:“David",
    dept: "IT"
}
```

Creating Objects

Using new operator from a class

```
class Employee {
  constructor() {
    this.name = '';
    this.dept = 'general';
  }
}

let emp = new Employee();
emp.name = 'David';
```

Using literal

```
let emp = {

    name:"David",
    dept: "IT"
}
```

Using functions

```
function createEmp (name, dept){
            return {
                        name: name,
                        dept: dept
            }
}

let emp = createEmp('David', 'IT');
```

Using Prototype with Object.create

```
const employee = {
            name: '',
            dept: ''
}

const emp = Object.create (employee);
emp.name = 'David';
emp.dept = 'IT';
```

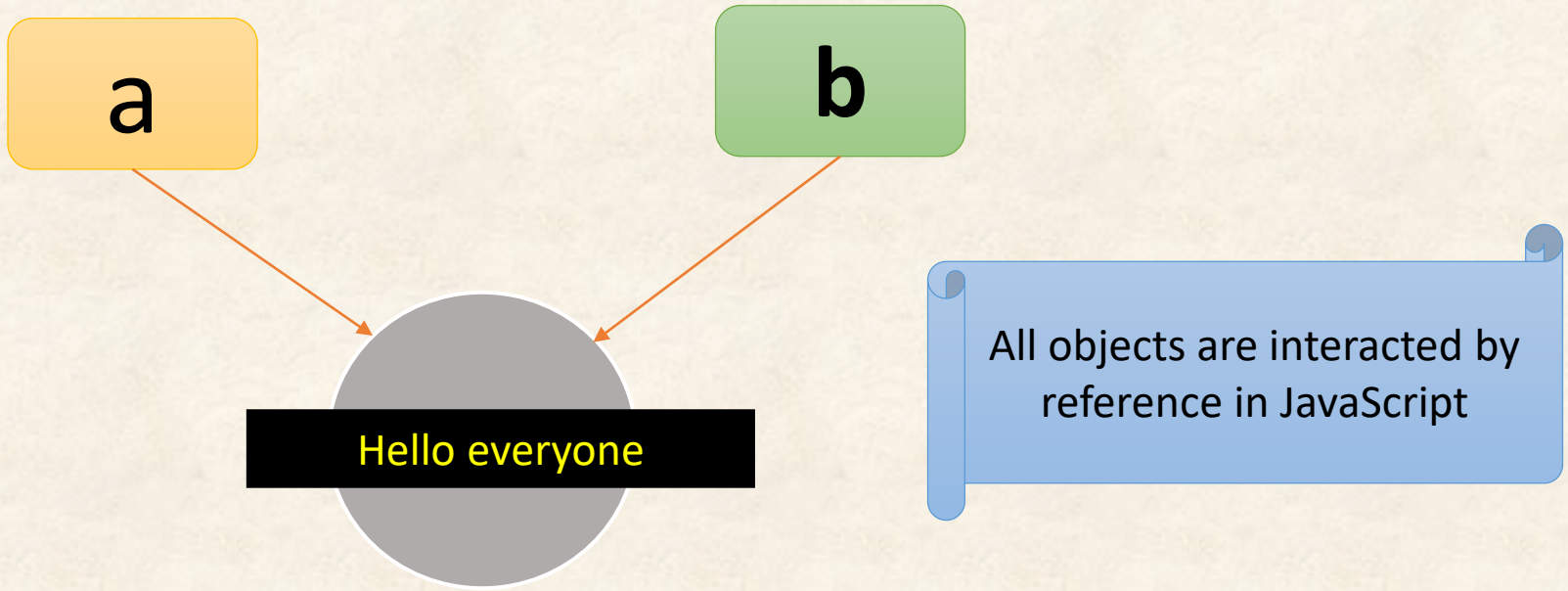
Experiment 1

Code Snippet

```
let a = {welcomeMsg: 'Welcome to this session'};
let b;
b = a;
a.welcomeMsg = 'Hello everyone';
console.log(b.welcomeMsg);
```

O/P

```
'Hello everyone'
```



All objects are interacted by reference in JavaScript

Defining & Using Properties

Key/value using semicolon

```
let emp = {

    name:"David",
    dept: "IT"
}

//to delete property
delete emp.name;
```

Assigning hardcoded Property

```
let emp = {

    name:"David",
    dept: "IT"
}
emp.Id = "1001";
```

Dynamic Assignment

```
emp [dynamicValue] = 'Kolkata';

emp ['id'] = 1001;
```

defineProperty (extra options)

```
Object.defineProperty(emp, 'doj',
{
     value: new Date(),
     writable: false
});
```

Writable: value of property can be changed
Enumerable: whether the property can be found using Object.keys()
Configurable: if property can be deleted / modified

Using getter/setter

```
let emp = {
  sname: '',
  get name(){
      return this.sname;
  },
  set name(str){
    this.sname = str;
   }
}
emp.name ='David';
```

Define an Object

```
let emp = {

    name:“David",
    dept: "IT"
}
```

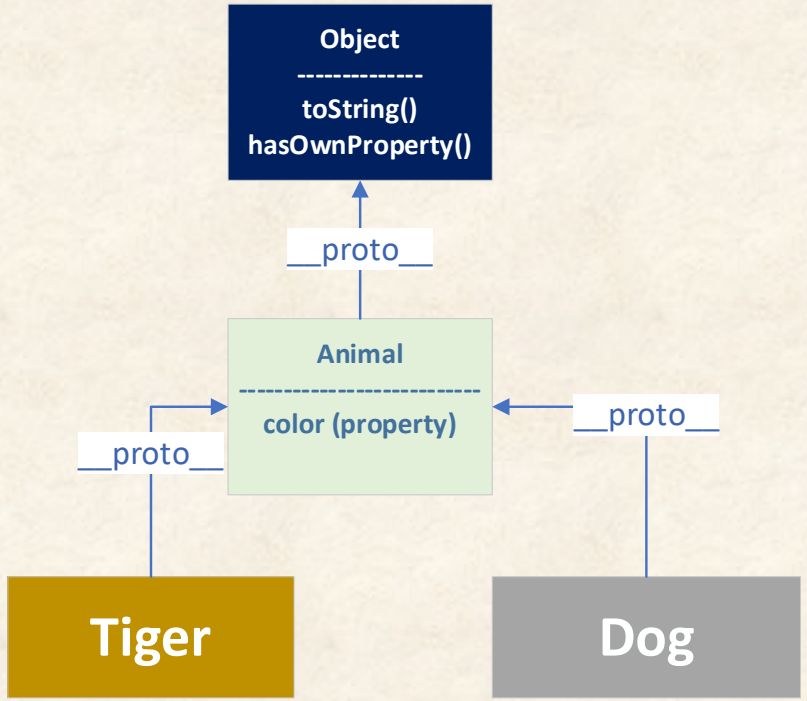
Object.keys and Object.values

```
console.log (Object.keys(emp)); // Array [“name”,“dept”]


console.log (Object.values(emp)); //Array [“David”,“IT”]
```

Inheritance

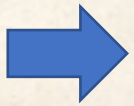
- We can inherit the properties/ methods from the parent class to child class by using ‘extends’ keyword.

  Here, class Tiger **extends** Animal {}



Arrow Function *(introduced in ES6)*

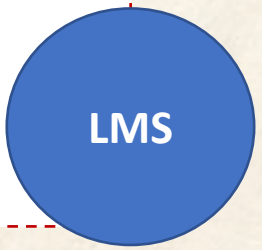
```
const squareANumber = function(number) {
        return number * number;
}
```



```
const squareANumber = (number) => number * number;
```

It inherits value of this from where they were called

```
import {subscribe, unsubscribe, APPLICATION_SCOPE, MessageContext} from 'lightning/messageService';
export default class DisplayLocationSubscriber extends LightningElement {
        subscribeToMessageChannel() {
                if (!this.subscription) {
                        this.subscription = subscribe(
                                this.messageContext,
                                selectedEntity,
                                (message) => this.handleMessage(message),
                                { scope: APPLICATION_SCOPE }
                        );
                }
        }

        handleMessage(message) {
                //do something
        }
}
```



Experiment 2 *(using arrow function)*

```
class MyClass {
  constructor(){
      this.str = new String();
      /*setTimeout( function() {
        this.str = this.greeting();
      },1000);*/

      setTimeout( () => {
        this.str = this.greeting();
      }, 1000);
  }

  greeting() {
    console.log('Hello everyone');
  }
}

let a = new MyClass();
```

O/P

```
'Hello everyone'
```

Browser and Events (17%)

Event Handling

defines the events/action initiation by user or browser

window events

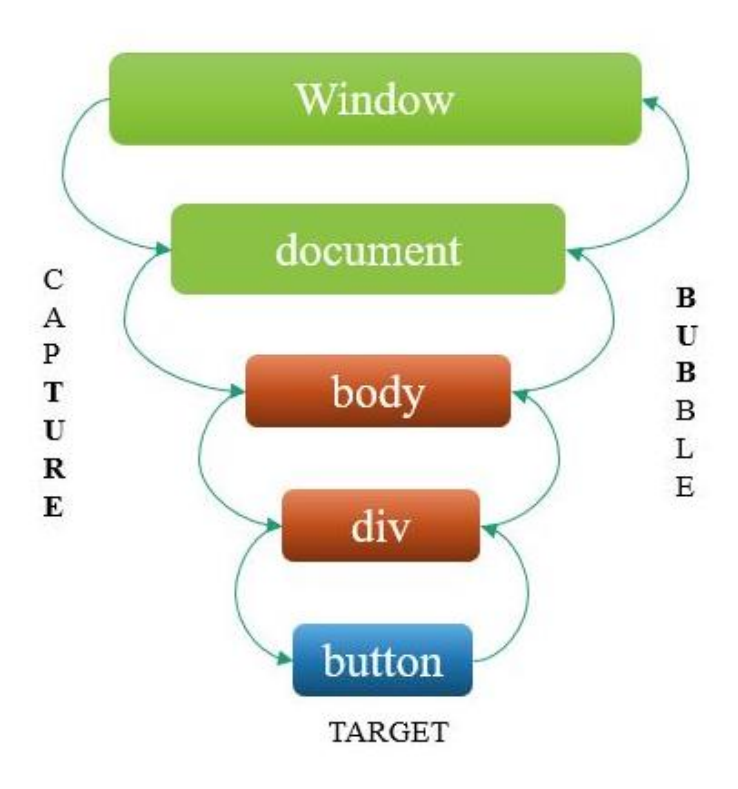
Statement	Functionalities
window.history.back();	It is used to redirect to the previous page.
window.history.forward();	It is used to forward the next page.
window.scrollTo(0, 1000);	It is used to scroll the particular point automatically.
window.location.href	It is used to navigate the specific URL.

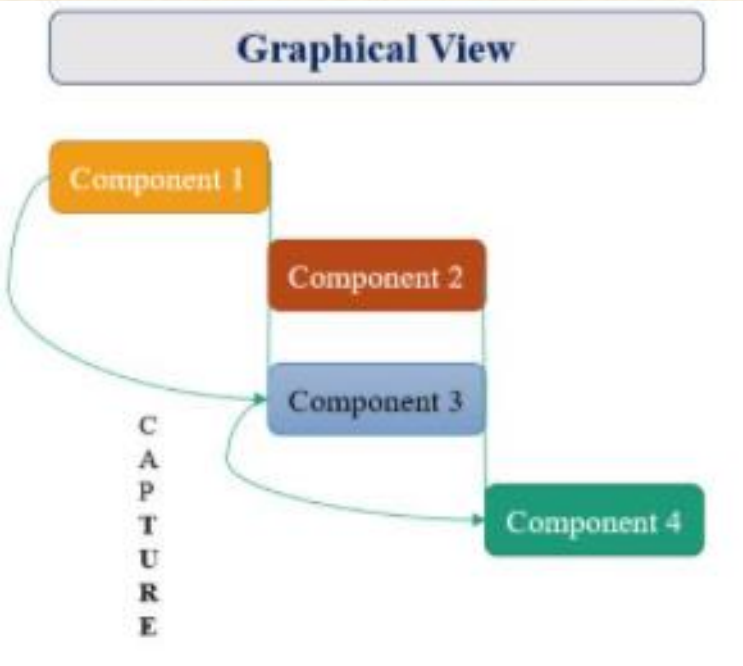
html events

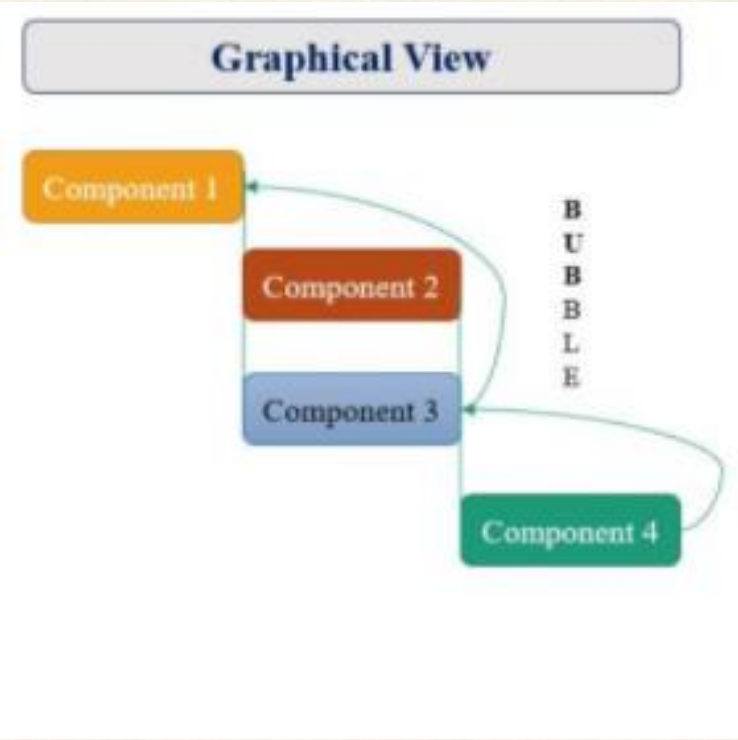
Statement	Functionalities
onchange	It will be triggered based on the element change
onclick	It will be triggered when user clicks on element
onload	It will be triggered for onloading page
onkeydown	It will be triggered on the pressing of key down, same is applicable for key up
onmouseover, onmouseout	It will be triggered on mouse over/ mouse away from the element.

Event Propagation

defines how to travel events through Document Object Model (DOM) tree.







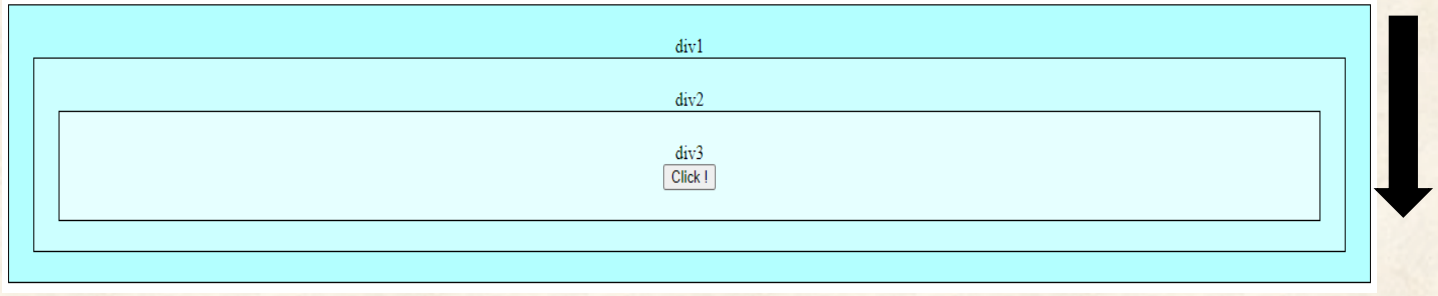
Capture – propagate from window to the target element through DOM tree
Target – event is generated from this element in the DOM tree
Bubble - propagate from target element to window through DOM tree

Experiment 1

Code Snippet

```
<body>
    <div id="div1" class="divBlock">
        div1
        <div id="div2" class="divBlock">
            div2
            <div id="div3" class="divBlock">
                div3
                <br/>
                <button id="btn" class="divBlock">Click !</button>
            </div>
        </div>
    </div>
    <script>
        function checkEvents() {
            console.log(this.getAttribute("id"));
        }
        const divBlocks = document.querySelectorAll(".divBlock");
        let capture = true;
        divBlocks.forEach(function (d) {
            d.addEventListener("click", checkEvents, capture);
        });
    </script>
</body>
```

Capture Phase



O/P

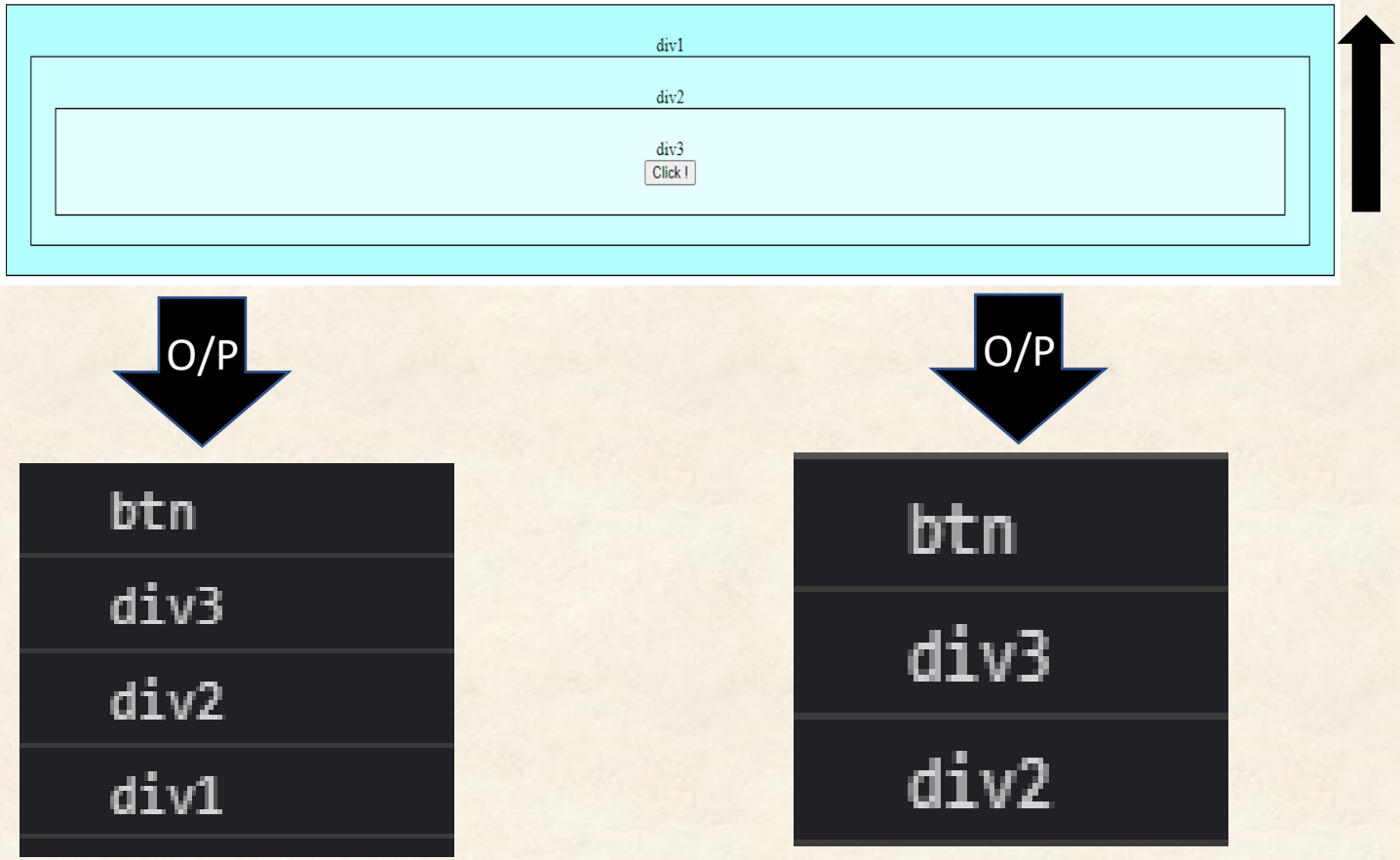
```
div1
div2
div3
btn
```

Experiment 2

Code Snippet

```
<body>
    <div id="div1" class="divBlock">
        div1
        <div id="div2" class="divBlock">
            div2
            <div id="div3" class="divBlock">
                div3
                <br/>
                <button id="btn" class="divBlock">Click !</button>
            </div>
        </div>
    </div>
    <script>
        function checkEvents() {
            console.log(this.getAttribute("id"));
        }
        const divBlocks = document.querySelectorAll(".divBlock");
        let capture = false;
        divBlocks.forEach(function (d) {
            d.addEventListener("click", checkEvents, capture);
        });
    </script>
</body>
```

Bubble Phase (default)



fire custom events

To trigger the custom events, we need to dispatch the event by following syntax:
this.dispatchEvent(new CustomEvent('eventName', detail:{parameters}))

event.stopPropagation() at div2

<<halts event propagation>>

Asynchronous Programming (13%)

Asynchronous Programming

JavaScript is

- **Synchronous**
- **blocking**
- **single-threaded language**

It means only one operation can be in progress at a time

```
function a() {
  console.log("*** I am in a ****");
  b();
}
function b() {
  console.log("*** I am in b ****");
}
a();
```

```
*** I am in a ****
*** I am in b ****
```

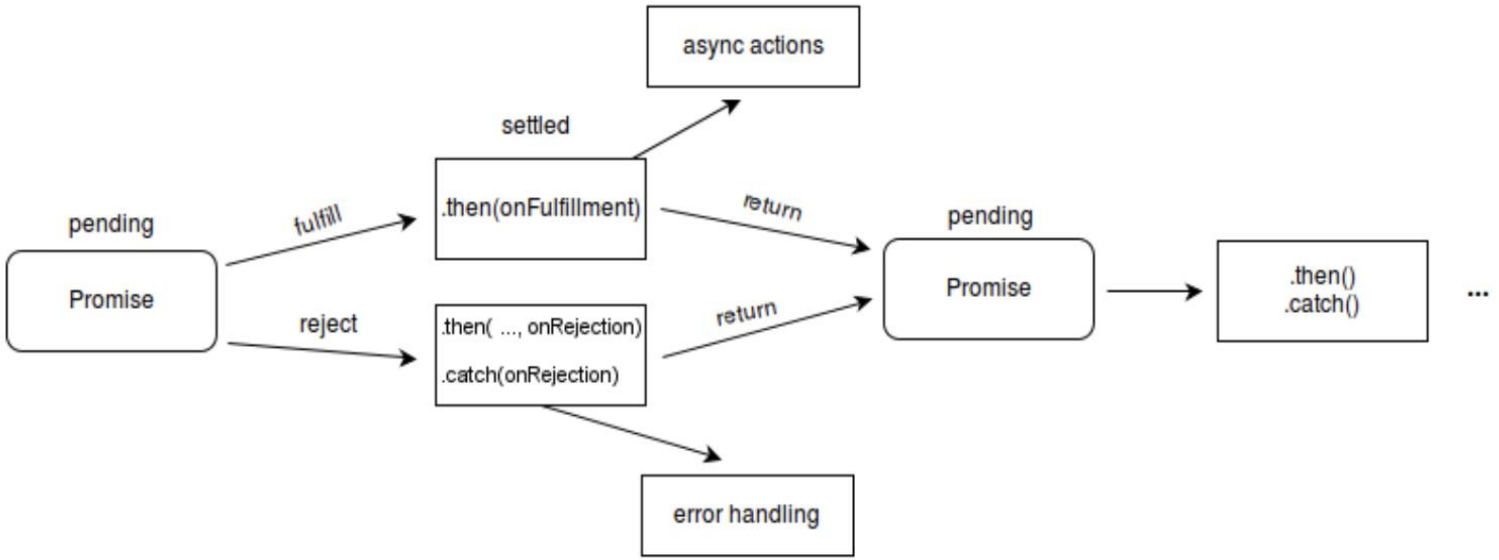
Asynchronous Execution (Promise/ async-await)

```
function foo() {
  console.log(" ******* in foo 1 ******* ");
  delayedFun().then(result => {
    console.log(" ******* return from delayed fun ******* " + result);
  });
  console.log(" ******* in foo 2 ******* ");
}
function delayedFun() {
  return new Promise((resolve, reject) => {
    setTimeout(function () {
      resolve("***** Hello from async *********");
    }, 3000);
  });
}
foo();
```

```
******* in foo 1 *******
******* in foo 2 *******
******* return from delayed fun ******* ***** Hello from async *********
```

A real-life coding scenario. Build an account 360 from different data sources.

- Number of cases open - Salesforce
- Number of Open Opportunity - Salesforce.
- Product information (External system)
- Payment Information based on product data (External system)




```
get360Data() {
    getSFData(this.recId).then((accDetail) => {
        // Update the UI with data from Salesforce
        getProductData(accDetail.extId__c).then((productDetails) => {
            // Update UI with product detail and call payment information
            getPaymentData(productDetails).then((paymentDetail) => {
                // Update UI with payment information
                this.allLoaded = true;
            })
        })
    })
}
```

Simplify the promise chain with async-await

```
get360Data() {
    getSFData(this.recId).then((accDetail) => {
        // Update the UI with data from Salesforce
        getProductData(accDetail.extId__c).then((productDetails) => {
            // Update UI with product detail and call payment information
            getPaymentData(productDetails).then((paymentDetail) => {
                // Update UI with payment information
                this.allLoaded = true;
            })
        })
    })
}
```

```
async get360Data() {
            let accDetail = await getSFData(this.recId);
            let productDetails = await getProductData(accDetail.extId__c);
            let paymentDetail = await getPaymentData(productDetails);

    }
}
```

Digging dipper into Promise

Promise has 3 states

- Pending – waiting for execution (Default)
- Resolve - Success
- Reject - Failure

1. **Promise.all (iterable)** : Wait for all promises to be resolved, or for any to be rejected.

```
const promise1 = Promise.resolve(3);
const promise2 = 42;
const promise3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([promise1, promise2, promise3]).then((values) => {
  console.log(values);
});
// expected output: Array [3, 42, "foo"]
```

Promise.allSettled (iterable) : method returns a promise that resolves after all of the given promises have either fulfilled or rejected, with an array of objects that each describes the outcome of each promise.

It is typically used when you have multiple asynchronous tasks that are not dependent on one another to complete successfully, or you'd always like to know the result of each promise.

```
const promise1 = Promise.resolve(3);
const promise2 = new Promise((resolve, reject) => setTimeout(reject, 100, 'foo'));
const promises = [promise1, promise2];

Promise.allSettled(promises).
  then((results) => results.forEach((result) => console.log(result.status)));

// expected output:
// "fulfilled"
// "rejected"
```

Promise.any (iterable): Takes an iterable of Promise objects and, as soon as one of the promises in the iterable fulfills, returns a single promise that resolves with the value from that promise.

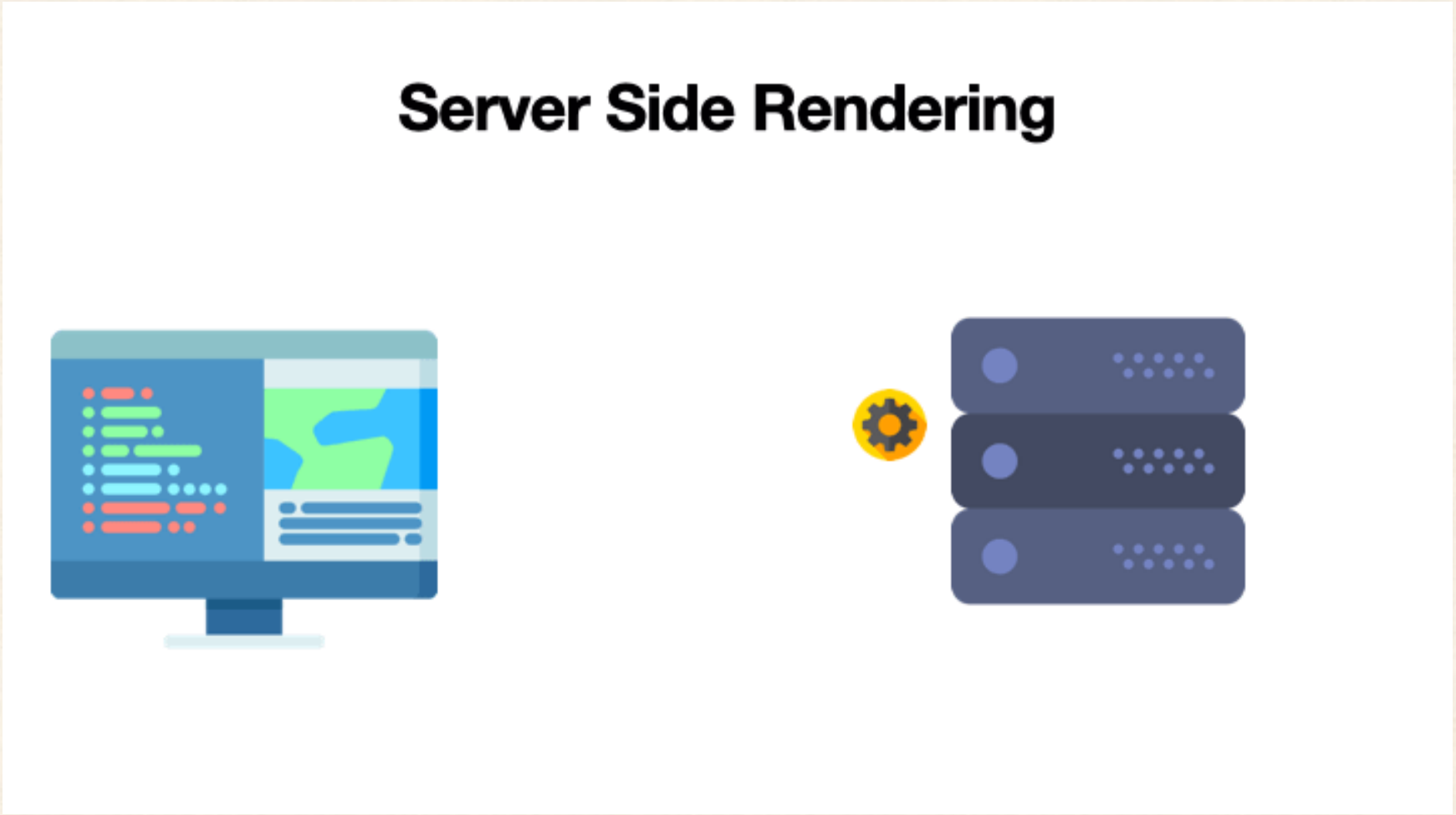
```
const promise1 = Promise.reject(0);
const promise2 = new Promise((resolve) => setTimeout(resolve, 100, 'quick'));
const promise3 = new Promise((resolve) => setTimeout(resolve, 500, 'slow'));

const promises = [promise1, promise2, promise3];

Promise.any(promises).then((value) => console.log(value));

// expected output: "quick"
```

Promise.race(iterable): Wait until any of the promises is resolved or rejected.

```
const promise1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 500, 'one');
});
const promise2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'two');
});
Promise.race([promise1, promise2]).then((value) => {
  console.log(value);
  // Both resolve, but promise2 is faster
});
// expected output: "two"
```

Server-Side JavaScript (8%)

Server Side Rendering

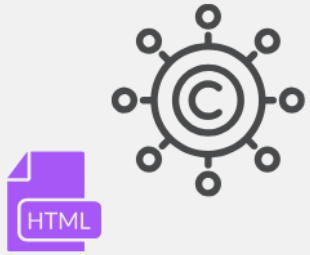


Basics – Client vs. Server-Side Rendering

1.

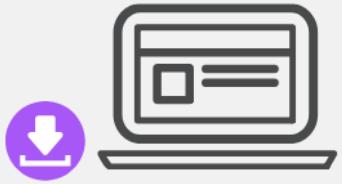
User Requests a Website

2.



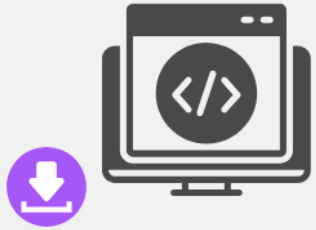
a CDN can quickly serve HTML files with links to JS

3.



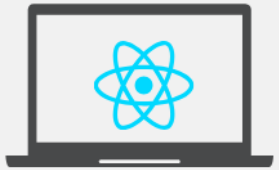
Browser downloads the HTML and then the JS, meanwhile the site isn't visible to the user

4.



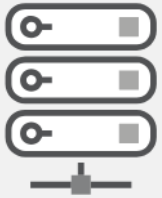
The Browser downloads the ıscripts

5.



The JS is then executed, APIs are called for data, & user sees placeholders

6.



The Server Responds with the data asked by the API

7.

The data from the APIs fill the placeholders and the page is now interactive

Basics – Client vs. Server-Side Rendering

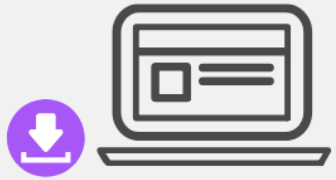
1.

User Requests a Website

2.

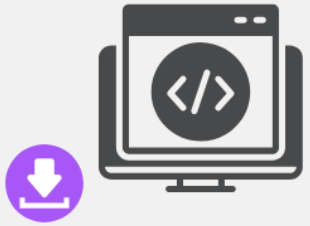
Server creates "Ready to Render" HTML files

3.



The Browser can quickly render the HTML but the site isn't interactive

4.

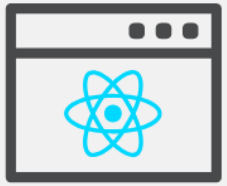


The Browser downloads the ascript

5.

The user can view content and the interactions can be recorded

6.



The Browser Executes the JS Framework

7.

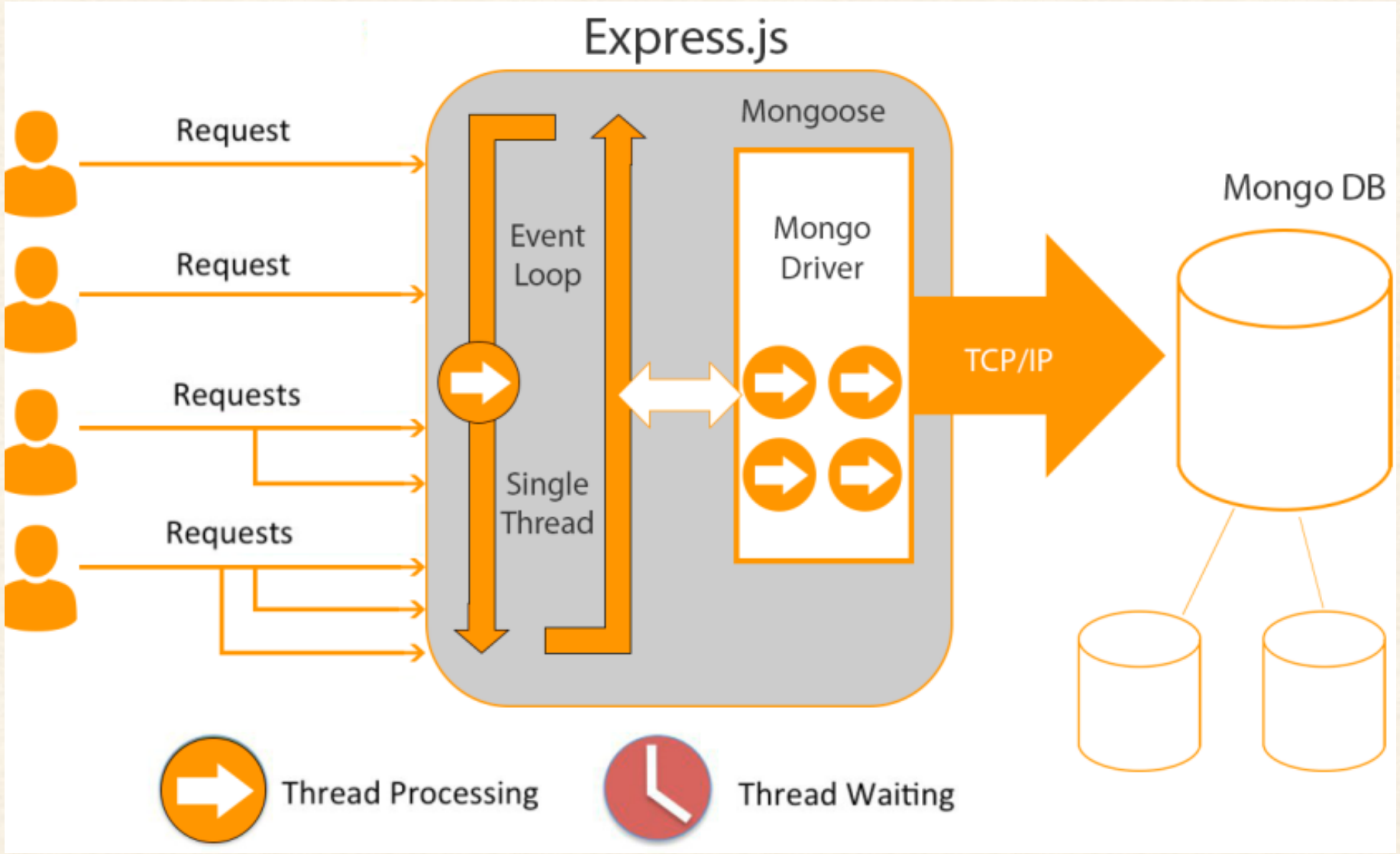
The recorded interactions can be executed and the page is now interactive

Node.js & Salesforce

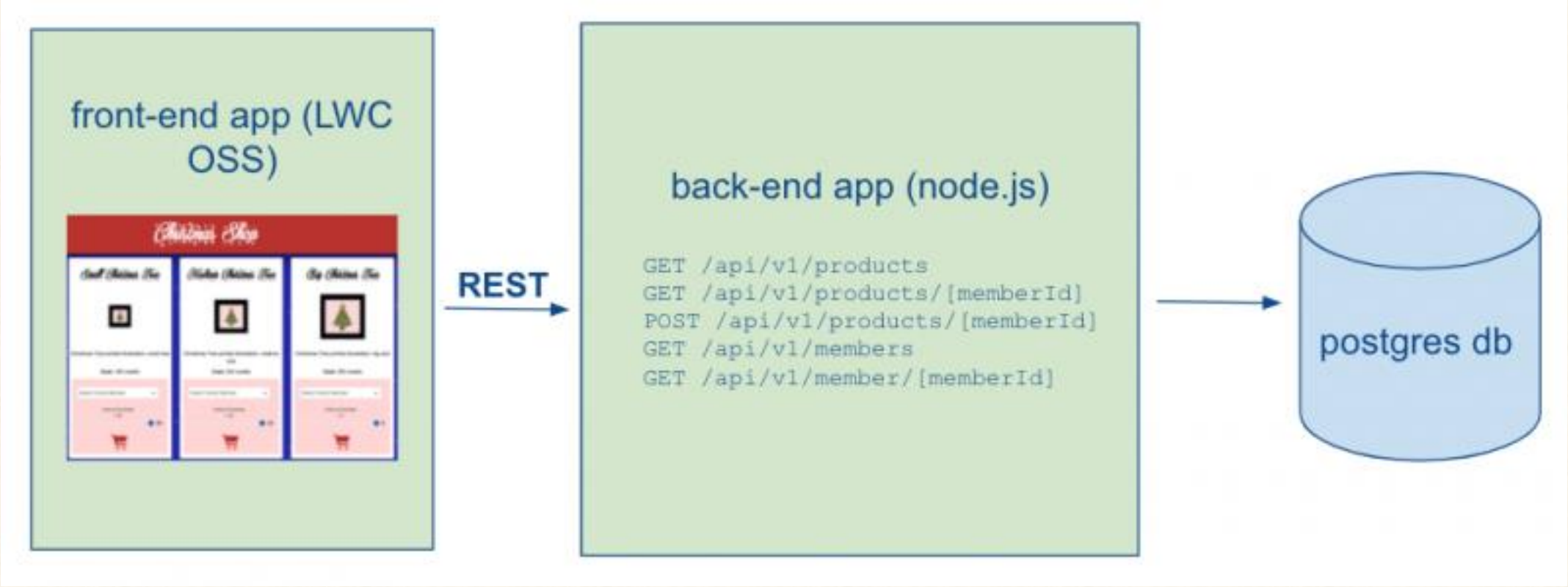
Nforce - https://github.com/kevinohara80/nforce

Jsforce - https://github.com/jsforce/jsforce

Node.js & Express

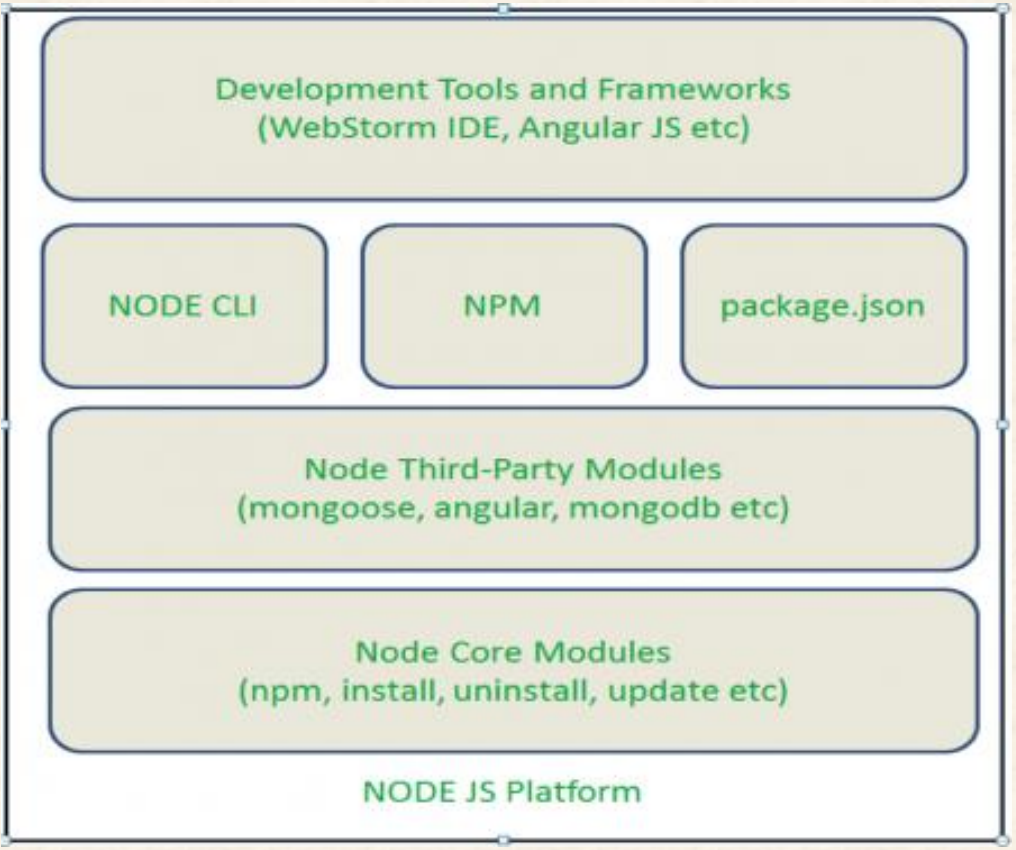


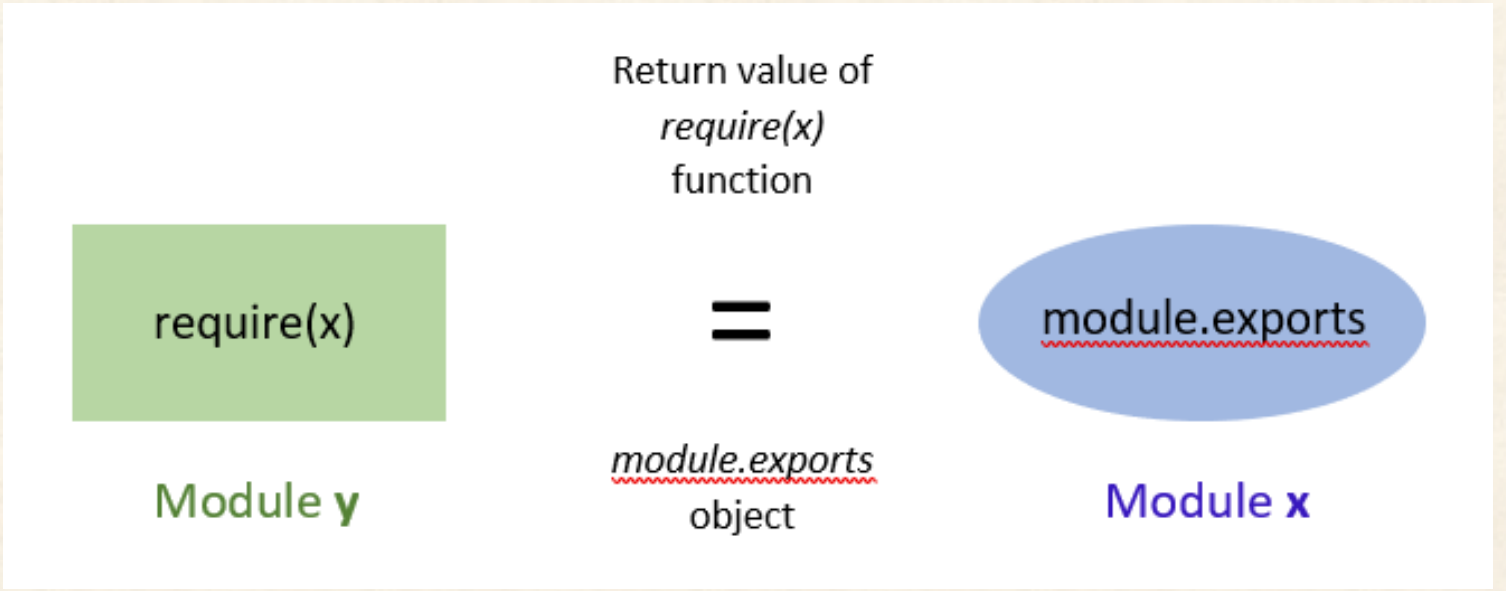
Node.js & LWC


front-end app (LWC OSS)
REST
back-end app (node.js)
GET /api/v1/products
GET /api/v1/products/[memberId]
POST /api/v1/products/[memberId]
GET /api/v1/members
GET /api/v1/member/[memberId]
postgres db

Node.js – Key HTTP Methods

Node.js – Key Concepts





Node.js – Packaging

NPM vs YARN

Created By
Gant Laborde of Infinite Red

What you need to know

```
npm install === yarn
```

Install is the default behavior.

```
npm install taco --save === yarn add taco
```

The Taco package is saved to your **package.json** immediately.

```
npm uninstall taco --save === yarn
remove taco
```

—**savecan** be defaulted in NPM by npm config set save true but this is non-obvious to most developers. Adding and removing from **package.json** is default in Yarn.

```
npm install taco --save-dev === yarn
add taco --dev
```

```
npm update --save === yarn upgrade
```

Great call on upgrade vs update, since that is exactly what it is doing! Version number moves, upgrade is happening!

WARNING **npm update --save** seems to be kinda broken in 3.11

```
npm install taco@latest --save === yarn
add taco
```

```
npm install taco --global === yarn global
add taco
```

As always, use global flag with care.

Things NPM has that yarn doesn't

`npm xmas`	**NO EQUIVALENT**
`npm visnup`	**NO EQUIVALENT**

What you know about Yarn

```
npm init === yarn init
```

```
npm link === yarn link
```

```
npm outdated === yarn outdated
```

```
npm publish === yarn publish
```

```
npm run === yarn run
```

```
npm cache clean === yarn cache clean
```

```
npm login === yarn login
```
(and logout)

```
npm test === yarn test
```

Things yarn has that NPM doesn't

```
yarn licenses ls
```

Allows you to inspect the licenses of your dependencies

```
yarn licenses generate-disclaimer
```

Automatically create your license dependency disclaimer

```
yarn why taco
```

Identify why 'taco' package is installed, detailing which other packages depend upon it (thanks Olivier Combe).

Debugging, Error Handling (7%)

Debugging

Every time, we need to debug code for searching errors and diagnose the code base to check the desired output

Statement	Description	When?
debugger;	It stops the execution of JavaScript and call the debugging function	You can use the debugger to see the output at the particular breakpoint
console.log()	It is used to print the result of any variables	If you want to show the output of variables, you can use
console.table()	It displays the data in a table format	You can printout the array of objects by this function to concise the output
console.warn()	It will show the warning message.	You can print out the error message from try-catch block to see the warning message
console.error()	It will show the error message.	You can print out the error message from try-catch block to see the error message
console.assert()	It helps to assertion the results. If assertion fails, this function will return an error message.	It's required at the time of assertion to check the stability of DOM element.
console.time() console.timeEnd()	It helps to determine the execution time of certain block.	You can print the timeframe to execute the certain block to check the system performance

Experiment 1

Code Snippet

```
console.log('Welcome to JavaScript Debugging');
console.time('Timer');
let agenda = [];

let group = {
  name: 'Salesforce Developer Group',
  city: 'Kolkata',
  country: 'India',
}
agenda = [group];

console.log(agenda);
console.table(agenda);

console.timeEnd('Timer');
```

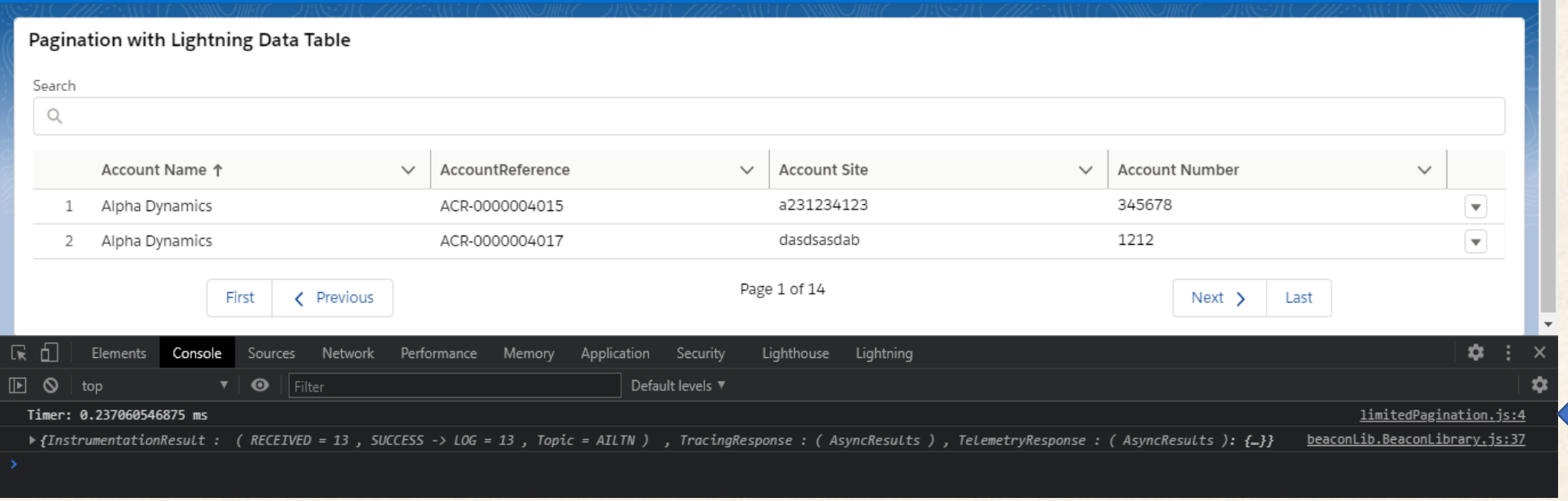
O/P

```
'Welcome to JavaScript Debugging'
[
  {
    name: 'Salesforce Developer Group',
    city: 'Kolkata',
    country: 'India'
  }
]
[
  {
    name: 'Salesforce Developer Group',
    city: 'Kolkata',
    country: 'India'
  }
]
| (index) |            name            |   city    | country |
-----------------------------------------------------------------
|    0    | 'Salesforce Developer Group' | 'Kolkata' | 'India' |

'Timer: 1ms'
```

Experiment 2 *(using console.time in LWC)*

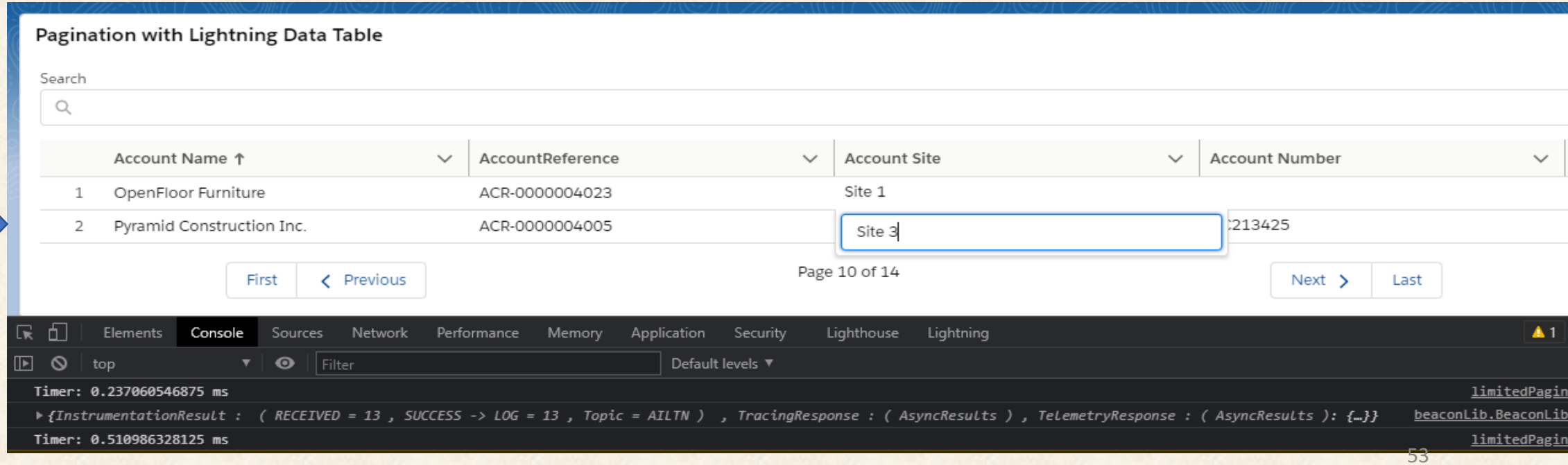
```
@wire(getLightningDataTableDetails, {sObjectAPI: 'Account', fieldSetName: 'AccountFieldSet', searchableFieldAPI: 'Name',
searchKey: '$searchKey', sortBy: '$sortedBy', sortDirection: '$sortedDirection'})
wiredAccounts(value) {
    console.time('Timer');
    this.wiredActivities = value;
    const { data} = value;
    this.columnsList= [];
    if (data) {
        this.items = data.sObjectList;
        if (this.items.length === 0){
            this.showTable = false;
        }else{
            this.totalRecordCount = this.items.length;
            this.totalPage = Math.ceil(this.totalRecordCount / this.pageSize);
            this.data = this.items.slice(0,this.pageSize);
            this.lastRecord = this.pageSize;
            data.fieldSetAttributesList.forEach(item=>{
                this.columnsList.push({
                    "label": item.label,
                    "fieldName": item.fieldName,
                    "type": item.type,
                    "editable": item.editable,
                    "sortable": item.sortable
                });

            })
        }
    }
    /** ............................. */
    /** ............................. */
    console.timeEnd('Timer');
}
```

Timer O/P



Onload

During Saving



Error Handling

Codes don’t go always well. Now, need to see how to manage the errors in JavaScript

- **try** statement to execute your code
- **catch** statement helps to handle the error
- **throw** statement helps to create custom errors
- **finally** statement always executes after try/catch block execution

```
try{
  //execute the code
  throw "error_message";
}catch(ex){
  //handle errors
}finally{
  //execute always
}
```

try without catch and without finally is possible in JS

Experiment 1

Code Snippet

```
try{
    //try to execute the code
    try{
      console.log('inner try');
    }catch(ex){
      console.log('inner catch');
      throw new Error('My Error Desc');
    }finally{
      console.log('inner finally');
    }
}catch(ex){
    console.log('outer catch');
}finally{
    console.log('outer finally');
}
```

O/P

```
'inner try'

'inner finally'

'outer finally'
```

Experiment 2

Code Snippet

```
try{
    //try to execute the code
    try{
      console.log('inner try');
      throw error;
    }catch(ex){
      console.log('inner catch');
      throw new Error('My Error Desc');
    }finally{
      console.log('inner finally');
    }
}catch(ex){
    console.log('outer catch');
}finally{
    console.log('outer finally');
}
```

O/P

```
'inner try'

'inner catch'

'inner finally'

'outer catch'

'outer finally'
```

Testing (8%)

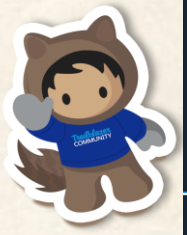
Testing

Unit Tests- Testing of individual units like functions or classes by supplying input and making sure the output is as expected:

expect(fn(5)).to.be(10)

Integration Tests- Testing processes across several units to achieve their goals, including their side effects:

```
const flyDroneButton = document.getElementById('fly-drone-button')

flyDroneButton.click()

assert(isDroneFlyingCommandSent())

//or even
drone.checkIfFlyingViaBluetooth()
  .then(isFlying => assert(isFlying))
```



Testing

End-to-end Tests (also known as e2e tests or functional tests)- Testing how scenarios function on the product itself, by controlling the browser or the website. These tests usually ignore the internal structure of the application entirety and look at them from the eyes of the user like on a black box.

Go to page "https://localhost:3303"

Type "test-user" in the field "#username"

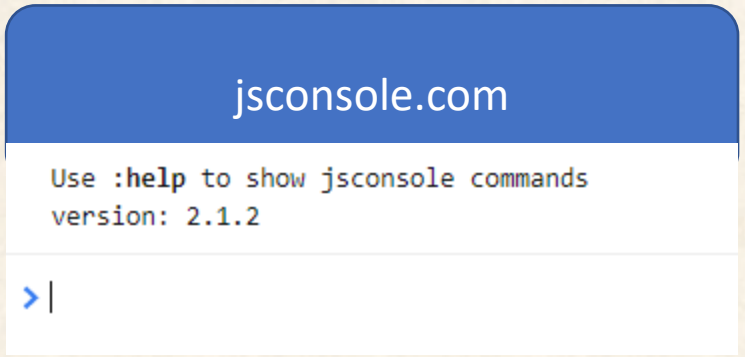
Type "test-pass" in the field "#password"

Click on "#login"

Expect Page Url to be https://localhost:3303/dashboard

Expect "#name" to be "test-name"

Tools to use


jsconsole.com
Use :help to show jsconsole commands
version: 2.1.2
>


JS

Recap

- We have learnt following topics today
  - Exam topics in JavaScript certifications
  - Sample Use Cases related to JavaScript
  - How JavaScript knowledge can be leveraged in LWC development

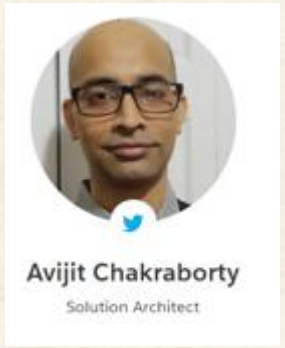
Q & A

Few Contributions from us

Keep in touch!

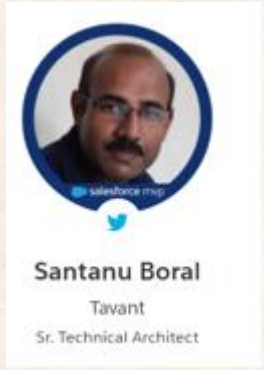
Website: http://www.gauravkheterpal.com/

@gauravkheterpal



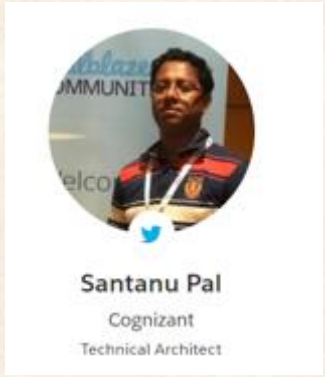
Website: https://avionsalesforce.blogspot.com/

@avi31c



Website: http://santanuboral.blogspot.com/

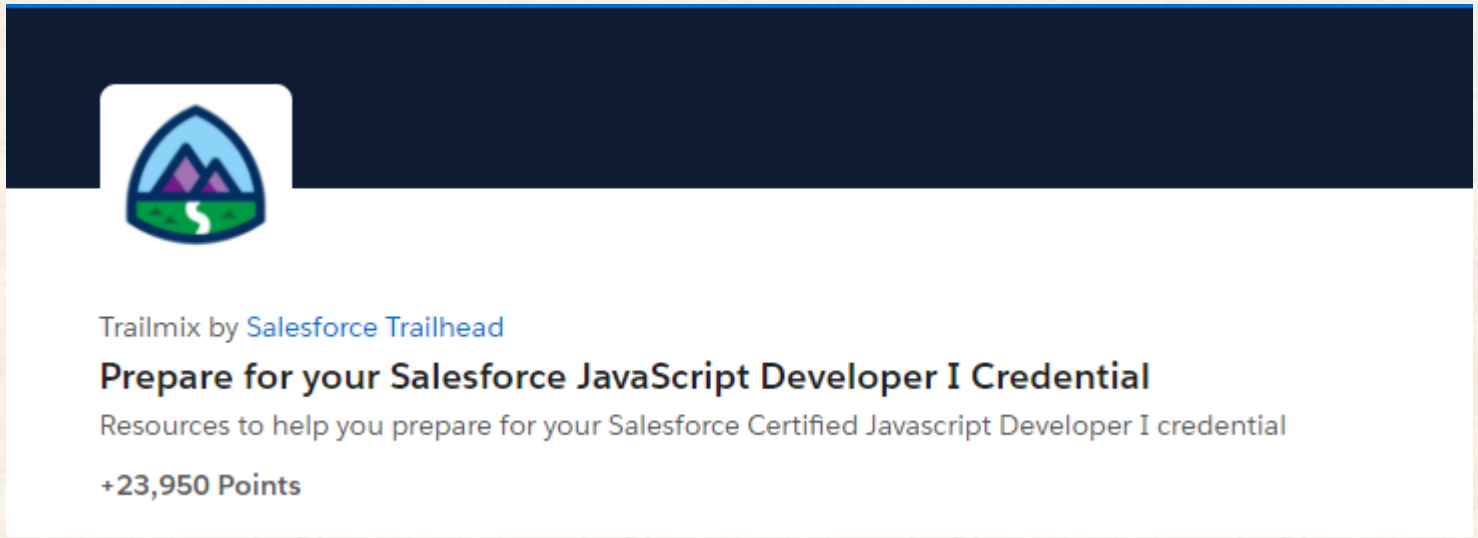
@santanuboral



Website: https://santanuatonline.com/

@drsantanupal

Further Study Reference



- Salesforce JavaScript Developer I Certification Cheat Sheet - 6 Pager
- Tips for passing Salesforce certified JavaScript Developer I Certification
- Salesforce JavaScript 1 certification Series by Nikhil Karkra

Thank You!

Salesforce Kolkata Developer Group