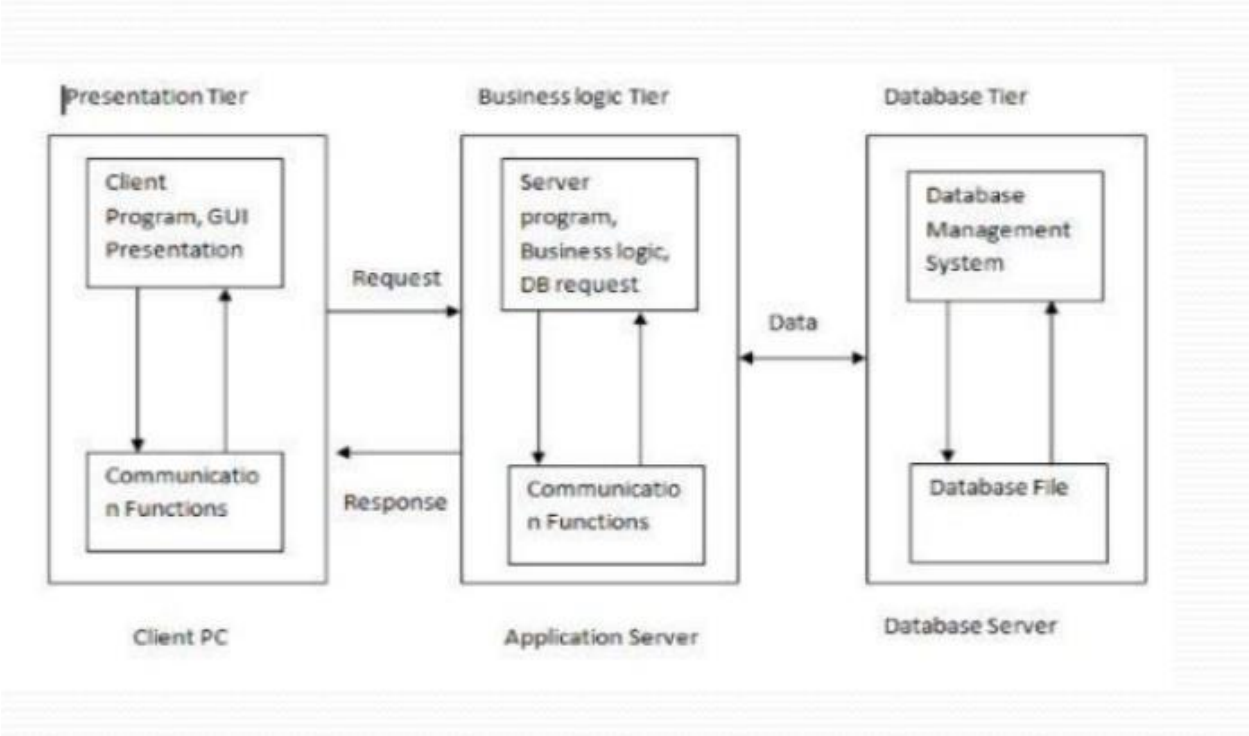
Sample Questions:

Module 1:

1. **The architecture of Web application.**



Typically a web-based application architecture comprises 3 core components:
1) Web Browser: The browser or the client-side component or the front-end component is the key component that interacts with the user, receives the input and manages the presentation logic while controlling user interactions with the application. User inputs are validated as well, if required.

2) Web Server: The web server also known as the backend component or the server-side component handles the business logic and processes the user requests by routing the requests to the right component and managing the entire application operations. It can run and oversee requests from a wide variety of clients.

3) Database Server:
The database server provides the required data for the Application. It handles data-related tasks. In a multi-tiered architecture, database servers can manage business logic with the help of stored procedures.

=> 3 TIER ARCHITECTURE
There are three layers of a 3-Tier architecture:
1. **Presentation layer / Client Layer**
The client-side component of a web application architecture enables users to interact with the server and the backend service via a browser.
- The code resides in the browser, receives requests and presents the user with the required information.

- This is where UI/UX design, dashboards, notifications, configurational settings, layout and interactive elements come into the picture.

2. **Application Layer / Business Layer**

The server-side component is the key component of the web application architecture that receives user requests, performs business logic and delivers the required data to the front-end systems.

- It contains servers, databases, web services etc.
- Web Server
- The web server uses HyperText Transfer Protocol (HTTP) along with other protocols to listen to user requests via a browser.
- It processes them by applying business logic and delivering the requested content to the end-user.

3. **Data Layer**

A database is a key component of a web application that stores and manages information for a web app.

- Using a function, you can search, filter and sort information based on user request and present the required info to the end user.
- They allow role-based access to maintain data integrity.
- While choosing a database for your architecture of web app, the size, speed, scalability and structure are the four aspects that require your consideration.
- For structured data, SQL-based databases are a good choice. It suits financial apps wherein data integrity is a key requirement.

2. HTTP methods

HTTP (Hypertext Transfer Protocol) specifies a collection of request methods to specify what action is to be performed on a particular resource. The most commonly used HTTP request methods are GET, POST, PUT, PATCH, and DELETE. These are equivalent to the CRUD operations (create, read, update, and delete).

GET: GET request is used to read/retrieve data from a web server. GET returns an HTTP status code of 200 (OK) if the data is successfully retrieved from the server.

POST: POST request is used to send data (file, form data, etc.) to the server. On successful creation, it returns an HTTP status code of 201.

PUT: A PUT request is used to modify the data on the server. It replaces the entire content at a particular location with data that is passed in the body payload. If there are no resources that match the request, it will generate one.

PATCH: PATCH is similar to PUT request, but the only difference is, it modifies a part of the data. It will only replace the content that you want to update.

DELETE: A DELETE request is used to delete the data on the server at a specified location.

3. **Explain GET and POST request methods of HTTP.**

HTTP GET	HTTP POST
In GET method we can not send large amount of data rather limited data is sent because the request parameter is appended into the URL.	In POST method large amount of data can be sent because the request parameter is appended into the body.
GET request is comparatively better than Post so it is used more than the Post request.	POST request is comparatively less better than Get so it is used less than the Get request.
GET request is comparatively less secure because the data is exposed in the URL bar.	POST request is comparatively more secure because the data is not exposed in the URL bar.
Request made through GET method are stored in Browser history.	Request made through POST method is not stored in Browser history.
GET method request can be saved as bookmark in browser.	POST method request can not be saved as bookmark in browser.
Request made through GET method are stored in cache memory of Browser.	Request made through POST method are not stored in cache memory of Browser.
Data passed through GET method can be easily stolen by attackers.	Data passed through POST method can not be easily stolen by attackers.
In GET method only ASCII characters are allowed.	In POST method all types of data is allowed.

4. **Differentiate following:**
   a. **HTTP and HTTPS**

- cant send large amount of data (bcoz request parameters are added in URL)
- Less secure(bcoz data is exposed in URL)[stolen by attackers]
- This request is stored in Browse History [cache memory](hence can be stored as bookmarks)
- Only ASCII characters are allowed

S.No.	HTTP	HTTPS
1.	HTTP stands for HyperText Transfer Protocol.	HTTPS for HyperText Transfer Protocol Secure.
2.	In HTTP, URL begins with “http://”.	In HTTPs, URL starts with “https://”.
3.	HTTP uses port number 80 for communication.	HTTPs uses 443 port number for communication.
4.	HTTP is considered to be unsecure.	HTTPs is considered as secure.
5.	HTTP works at Application Layer.	HTTPS works at Transport Layer.
6.	In HTTP, Encryption is absent.	Encryption is present in HTTPS.
7.	HTTP does not require any certificates.	HTTPS needs SSL Certificates.
8.	HTTP does not improve search ranking	HTTPS helps to improve search ranking
9.	HTTP faster than HTTPS	HTTPS slower than HTTP
10.	HTTP does not use data hashtags to secure data.	While HTTPS will have the data before sending it and return it to its original state on the receiver side.
11.	In HTTP Data is transfer in plaintext.	In HTTPS Data transfer in ciphertext.
12.	HTTP Should be avoided.	HTTPS Should be preferred.
13.	Search engines do not favour the insecure website.	Improved reputation of the website in search engine.
14.	HTTP Does not require SSL/TLS or Certificates	HTTPS Requires SSL/TLS implementation with Certificates.
15.	In HTTP Users ar worried about their data.	In HTTPS Users are confident about the security of their data.

b. SSL and TLS

SSL	TLS
SSL stands for Secure Socket Layer.	TLS stands for Transport Layer Security.
SSL (Secure Socket Layer) supports the **Fortezza** algorithm.	TLS (Transport Layer Security) does not support the **Fortezza** algorithm.
SSL (Secure Socket Layer) is the 3.0 version.	TLS (Transport Layer Security) is the 1.0 version.
In SSL(Secure Socket Layer), the Message digest is used to create a master secret.	In TLS(Transport Layer Security), a Pseudo-random function is used to create a master secret.
In SSL(Secure Socket Layer), the Message Authentication Code protocol is used.	In TLS(Transport Layer Security), Hashed Message Authentication Code protocol is used.
SSL (Secure Socket Layer) is more complex than TLS(Transport Layer Security).	TLS (Transport Layer Security) is simple.
SSL (Secure Socket Layer) is less secured as compared to TLS(Transport Layer Security).	TLS (Transport Layer Security) provides high security.
SSL is less reliable and slower.	TLS is highly reliable and upgraded. It provides less latency.
SSL has been depreciated.	TLS is still widely used.
SSL uses port to set up explicit connection.	TLS uses protocol to set up implicit connection.

c. XML and JSON

Summary of differences: JSON vs. XML

	JSON	XML
Stands for	*JSON* means JavaScript Object Notation.	*XML* means Extensible Markup Language.
History	Douglas Crockford and Chip Morningstar released JSON in 2001.	The XML Working Group released XML in 1998.
Format	JSON uses a maplike structure with key-value pairs.	XML stores data in a tree structure with namespaces for different data categories.
Syntax	The syntax of JSON is more compact and easier to read and write.	The syntax of XML substitutes some characters for entity references, making it more verbose.
Parsing	You can parse JSON with a standard JavaScript function.	You need to parse XML with an XML parser.
Schema documentation	JSON is simple and more flexible.	XML is complex and less flexible.
Data types	JSON supports numbers, objects, strings, and Boolean arrays.	XML supports all JSON data types and additional types like Boolean, dates, images, and namespaces.
Ease of use	JSON has smaller file sizes and faster data transmission.	XML tag structure is more complex to write and read and results in bulky files.
Security	JSON is safer than XML.	You should turn off DTD when working with XML to mitigate potential security risks.

d. URI, URL, URN

No	URI	URL	URN
1	URI stands for Uniform Resource Identifier	URL stands for Uniform Resource Location	URN stands for Uniform Resource Name
2	URI is a superset of URL & URN	URL is a subset of the Uniform Resource	URN is a subset of the Uniform Resource.
3	It used to identify a resource on the internet either by location or a name or both	It is used to identify a resource on the internet either by location	It uniquely identifies the resource by name
4	URI is not always a URL	All URLs are URIs	All URNs are URIs
5	URI includes components like scheme, authority, path, query, etc.	URL includes protocol,domain,path,hash,query,string etc	URN does not include any component
6	Example: https://www.geeksforgeeks.org/setting-environment-java/?ref=lbp	Example: https%3A%2F%2Fwww.geeksforgeeks.org%2Fsetting-environment-java%2F%3Fref%3Dlbp	Example: setting up the environment in java

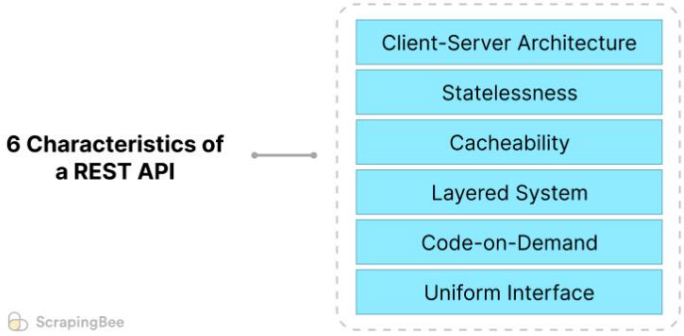
5. Short Note:

a. REST API and its characteristics

Representational **S**tate **T**ransfer (REST) is an architectural style that defines a set of constraints to be used for creating web services. **REST API** is a way of accessing web services in a simple and flexible way without having any processing.

REST (Representational State Transfer) is an architectural style for designing networked applications. RESTful APIs (Representational State Transfer Application Programming Interfaces) are a common implementation of REST principles in web services. REST APIs have several key characteristics that make them widely used and popular in web development.

Six Characteristics of REST



Client-Server architecture

RESTful APIs are built with a client-server architecture, meaning that the client sends a request to the server and the server sends back a response. The client-server architecture in REST emphasizes the separation of concerns, promoting scalability, loose coupling, and flexibility. This separation allows for independent development and maintenance of client and server components, making it a fundamental concept in building modern web applications and services. It enables effective resource management, data exchange, and collaboration across a distributed networked environment.

Statelessness

RESTful APIs are stateless, meaning that each request made by the client to the server contains all the information necessary for the server to fulfill the request, without relying on any previous requests or server-side storage.

Cacheability

It is important to utilize methods to reduce the load on the server. Therefore, RESTful APIs implement some sort of caching.

Layered System

REST requires the APIs to be designed as a layered system, where the client interacts with the server through a single endpoint, while the server can interact with multiple backend systems.

Code-on-Demand

This characteristic means that the server can send back code to be executed by the client instead of data.

It allows servers to extend the functionality of clients by transferring executable code to the client. In practice, this feature is not commonly used in most RESTful services. However, it can be useful in scenarios where clients are expected to execute code delivered from the server, such as in web browsers executing JavaScript code.

Uniform Interface

This means that the API uses a common set of methods, such as GET, POST, PUT, and DELETE, to access resources, and a standard format, such as JSON or XML, for requests and responses.

- Resource Identification: Resources are identified using URLs, which provide a globally unique identifier for each resource.
- Resource Manipulation: Resources can be manipulated using a small set of well-defined methods, such as GET (retrieve), POST (create), PUT (update), DELETE (remove), and more.
- Self-Descriptive Messages: Responses from the server include information about how to interpret the data (e.g., the media type specified in the Content-Type header). This allows clients to understand how to process the data without prior knowledge.
- Hypermedia as the Engine of Application State (HATEOAS): The server provides hyperlinks within the response, allowing clients to navigate the application's capabilities without prior knowledge of API endpoints. This enables more flexible and dynamic interaction between clients and servers.

b. DNS

An application layer protocol defines how the application processes running on different systems pass the messages to each other.

- DNS stands for Domain Name System.
- DNS is a directory service that provides a mapping between the name of a host on the network and its numerical address.
- DNS is required for the functioning of the internet.
- Each node in a tree has a domain name, and a full domain name is a sequence of symbols specified by dots.
- DNS is a service that translates the domain name into IP addresses. This allows the users of networks to utilize user-friendly names when looking for other hosts instead of remembering the IP addresses.
- For example, suppose the FTP site at EduSoft had an IP address of 132.147.165.50, most people would reach this site by specifying ftp.EduSoft.com. Therefore, the domain name is more reliable than the IP address.

6. **Structure of browser (components of the browser)**
7. **Functions of different components of the browser.**

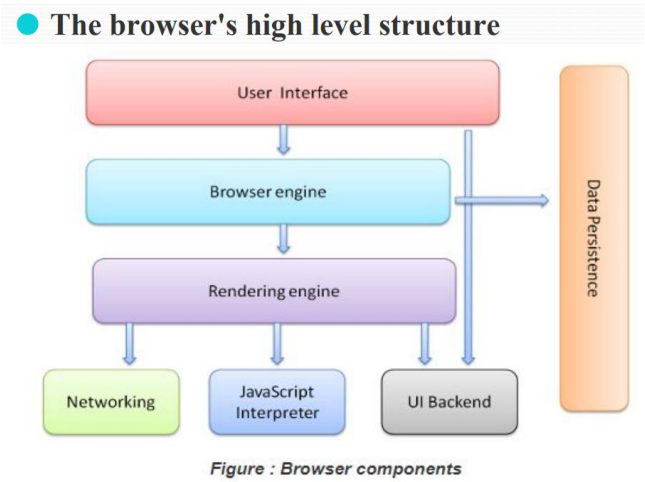


Figure : Browser components

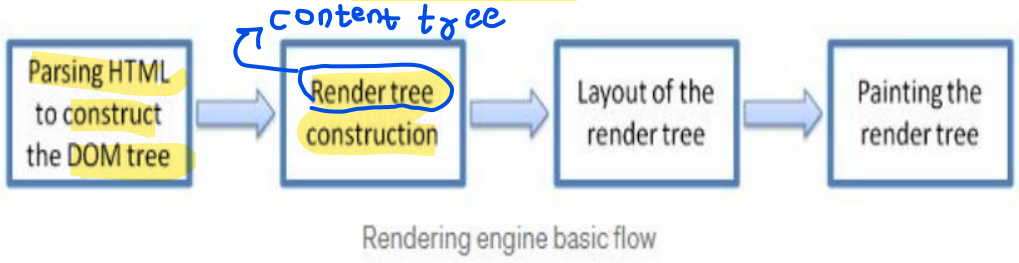
The browser's main components are :

1. **The user interface**: The user interface is the space where User interacts with the browser. It includes the address bar, back and next buttons, home button, refresh and stop, bookmark option, etc. Every other part, except the window where the requested web page is displayed, comes under it. This component allows end-users to interact with all visual elements available on the web page. The visual elements include the address bar, home button, next button, and all other elements that fetch and display the web page requested by the end-user.
2. **The browser engine**: It is a core component of every web browser. The browser engine functions as an intermediary or a bridge between the user interface and the rendering engine. It queries and handles the rendering engine as per the inputs received from the user interface.
3. **The rendering engine** : The responsibility of the rendering engine is to display the requested contents on the browser screen. The rendering engine, as the name suggests, is responsible for rendering the requested web page on the browser screen. The rendering engine interprets the HTML, XML documents and images that are formatted using CSS and generates the layout that is displayed in the User Interface.
4. **Networking**: For network calls such as HTTP requests, using different implementations for different platforms behind a platform-independent interface. This component is responsible for managing network calls using standard protocols like HTTP or FTP. It also looks after security issues associated with internet communication. Component of the browser which retrieves the URLs using the common internet protocols of HTTP or FTP. The networking component handles all aspects of Internet communication and security.
5. **UI backend**: Used for drawing basic widgets like combo boxes and windows. This backend exposes a generic interface that is not platform specific.
6. **JavaScript interpreter**: Used to parse and execute JavaScript code. It is the component of the browser which interprets and executes the javascript code embedded in a website. The interpreted results are sent to the rendering engine for display. If the script is external then first the resource is fetched from the network. Parser keeps on hold until the script is executed. As the name suggests,

it is responsible for parsing and executing the JavaScript code embedded in a website. Once the interpreted results are generated, they are forwarded to the rendering engine for displaying on the user interface.

7. **Data storage**: This is a persistence layer. The browser may need to save all sorts of data locally, such as cookies. Browsers also support storage mechanisms such as localStorage, IndexedDB, WebSQL and FileSystem.

8. **Working of the rendering engine**
   The networking layer will start sending the contents of the requested documents to the rendering engine in chunks of 8KBs.



Rendering engine basic flow

1. The rendering engine parses the chunks of HTML document and convert the elements to DOM nodes in a tree called the “content tree” or the “DOM tree”. It also parses both the external CSS files as well in style elements.
2. The rendering engine parses the chunks of HTML document and convert the elements to DOM nodes in a tree called the “content tree” or the “DOM tree”.
3. It also parses both the external CSS files as well in style elements. While the DOM tree is being constructed, the browser constructs another tree, the render tree. This tree is of visual elements in the order in which they will be displayed. It is the visual representation of the document.
4. The purpose of this tree is to enable painting the contents in their correct order. Firefox calls the elements in the render tree “frames”. WebKit uses the term renderer or render object.
5. After the construction of the render tree, it goes through a “layout process” of the render tree.
6. When the renderer is created and added to the tree, it does not have a position and size. The process of calculating these values is called layout or reflow.
7. This means giving each node the exact coordinates where it should appear on the screen.
8. The position of the root renderer is 0,0 and its dimensions are the viewport–the visible part of the browser window.
9. All renderers have a “layout” or “reflow” method, each renderer invokes the layout method of its children that need layout.
10. The next stage is painting. In the painting stage, the render tree is traversed and the renderer’s “paint()” method is called to display content on the screen. Painting uses the UI backend layer.

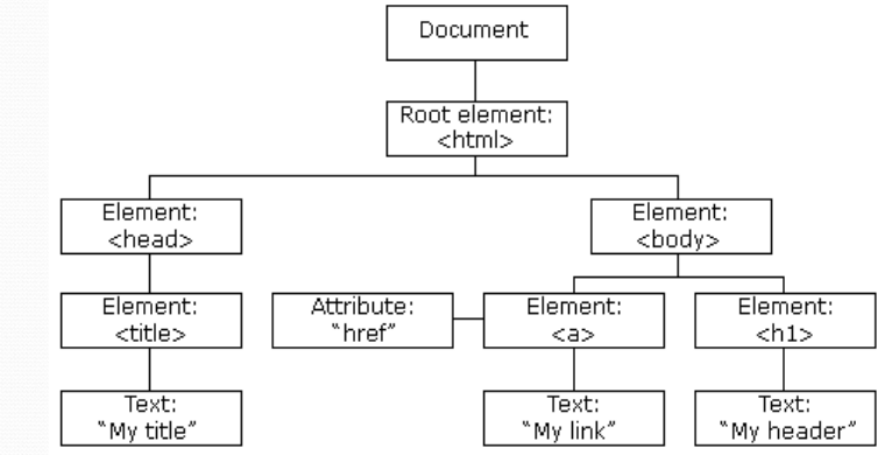
9. What is DNS? Explain domain and sub-domain with suitable examples.

An application layer protocol defines how the application processes running on different systems pass the messages to each other.

- DNS stands for Domain Name System.
- DNS is a directory service that provides a mapping between the name of a host on the network and its numerical address.
- DNS is required for the functioning of the internet.
- Each node in a tree has a domain name, and a full domain name is a sequence of symbols specified by dots.
- DNS is a service that translates the domain name into IP addresses. This allows the users of networks to utilize user-friendly names when looking for other hosts instead of remembering the IP addresses.
- For example, suppose the FTP site at EduSoft had an IP address of 132.147.165.50, most people would reach this site by specifying ftp.EduSoft.com. Therefore, the domain name is more reliable than IP address.
- **Domain**: A domain is a unique and human-readable name that identifies a specific location on the internet. It consists of two parts: the top-level domain (TLD) and the second-level domain (SLD). For example, in "example.com," "example" is the SLD, and ".com" is the TLD. Domains are used to represent websites, services, or resources on the internet.
- Example:
- Domain: "google.com"
- IP Address: 172.217.168.78
- **Sub-Domain**: A sub-domain is a part of a larger domain. It allows for the organization of websites or services under the main domain. Sub-domains are created by adding a prefix to the main domain name and are separated by dots. Each sub-domain can have its content or point to a different resource.
- Example:
- Main Domain: "blog.example.com"
- Sub-Domain: "blog"
- In this example, "blog" is a sub-domain of the main domain "example.com." It could be used to host a blog section of the website, and the full URL would be "blog.example.com."

10. Java Script DOM and different methods of Document object.

- *The HTML DOM views a HTML document as a tree-structure. The tree structure is called a node-tree.*



1) **getElementById(id)**: This method returns the element with the specified id attribute. It is one of the most commonly used methods for selecting elements by their unique identifier.

```
const element = document.getElementById("myElement");
```

2) **getElementsByClassName(className)**: This method returns a collection of elements that have the specified class name.

```
const elements = document.getElementsByClassName("myClass");
```

3) **getElementsByTagName(tagName):** This method returns a collection of elements with the specified tag name (e.g., "div", "p").

```
const elements = document.getElementsByTagName("div");
```

4) **querySelector(selector):** This method returns the first element that matches the specified CSS selector. It is versatile and allows you to select elements using complex selectors.

```
const element = document.querySelector("#myId .myClass");
```

5) **querySelectorAll(selector):** Similar to querySelector, this method returns a NodeList containing all elements that match the specified CSS selector.

```
const elements = document.querySelectorAll(".myClass");
```

6) **createElement(tagName):** This method creates a new HTML element with the specified tag name. You can then manipulate and append this element to the DOM.

```
const newDiv = document.createElement("div");
```

7) **appendChild(node):** This method appends a child node to an element. You can use it to add new elements to an existing element.

```
const parentElement = document.getElementById("parent");
parentElement.appendChild(newDiv);
```

8) **removeChild(node):** This method removes a child node from an element.

```
const parentElement = document.getElementById("parent");
const childElement = document.getElementById("child");
parentElement.removeChild(childElement);
```

9) **addEventListener(event, callback):** This method allows you to attach event listeners to elements. When the specified event occurs on the element, the provided callback function is executed.

```
const button = document.getElementById("myButton");
button.addEventListener("click", function() {
  // Your code here
});
```

10) **getElementById(id).innerHTML**: This property allows you to get or set the HTML content of an element with the specified id.

```
const element = document.getElementById("myElement");
const htmlContent = element.innerHTML;
```

11) **getElementById(id).textContent**: This property allows you to get or set the text content of an element with the specified id.

```
const element = document.getElementById("myElement");
      const textContent = element.textContent;
```

Module 2: Java Script

ES5	ES6
ECMA script is a trademarked scripting language specification defined by Ecma international. The fifth edition of the same is known as ES5	ECMA script is a trademarked scripting language specification defined by Ecma international. The sixth edition of the same is known as ES6
It was introduced in 2009.	It was introduced in 2015.
It supports primitive data types that are string, number, boolean, null, and undefined.	In ES6, there are some additions to JavaScript data types. It introduced a new primitive data type 'symbol' for supporting unique values.
There are only one way to define the variables by using the var keyword.	There are two new ways to define variables that are let and const.
It has a lower performance as compared to ES6.	It has a higher performance than ES5.
Object manipulation is time-consuming in ES5.	Object manipulation is less time-consuming in ES6.
In ES5, both function and return keywords are used to define a function.	An arrow function is a new feature introduced in ES6 by which we don't require the function keyword to define the function.
It provides a larger range of community supports than that of ES6	It provides a less range of community supports than that of ES5

edition

year

data types:- added was "symbol"

ways to define varaible

performance

time-consuming

function:- arrow

community support

Client-side scripting	Server-side scripting
Source code is visible to the user.	Source code is not visible to the user because its output of server-sideside is an HTML page.
Its main function is to provide the requested output to the end user.	Its primary function is to manipulate and provide access to the respective database as per the request.
It usually depends on the browser and its version.	In this any server-side technology can be used and it does not depend on the client.
It runs on the user's computer.	It runs on the webserver.
There are many advantages linked with this like faster response times, a more interactive application.	The primary advantage is its ability to highly customize, response requirements, access rights based on user.
It does not provide security for data.	It provides more security for data.
It is a technique used in web development in which scripts run on the client's browser.	It is a technique that uses scripts on the webserver to produce a response that is customized for each client's request.
HTML, CSS, and javascript are used.	PHP, Python, Java, Ruby are used.
No need of interaction with the server.	It is all about interacting with the servers.
It reduces load on processing unit of the server.	It surge the processing load on the server.

1. **Explain alert(), confirm() and prompt() method of window object.**
   The alert(), confirm(), and prompt() methods are commonly used in web development to interact with users by displaying dialog boxes in a web page. These methods are part of the window object in JavaScript and are used to

gather information from users or provide them with important messages. Here's an explanation of each of these methods:

1) **alert() -** An alert box is often used if you want to make sure information comes through to the user. When an alert box pops up, the user will have to click "OK" to proceed.
   Syntax - window.alert("sometext");
   Eg-alert("I am an alert box!");
2) **confirm() -** A confirm box is often used if you want the user to verify or accept something. When a confirm box pops up, the user will have to click either "OK" or "Cancel" to proceed. If the user clicks "OK", the box returns true. If the user clicks "Cancel", the box returns false.
   Syntax- window.confirm("sometext");
   Eg-if (confirm("Press a button!")) {
   txt = "You pressed OK!";
   } else {
   txt = "You pressed Cancel!";
   }
3) **prompt()** - A prompt box is often used if you want the user to input a value before entering a page.When a prompt box pops up, the user will have to click either "OK" or "Cancel" to proceed after entering an input value. If the user clicks "OK" the box returns the input value. If the user clicks "Cancel" the box returns null.
   Syntax- window.prompt("sometext","defaultText");
   Eg- let person = prompt("Please enter your name", "Harry Potter");
   let text;
   if (person == null || person == "") {
   text = "User cancelled the prompt.";
   } else {
   text = "Hello " + person + "! How are you today?";
   }

2. **Explain output and input methods in Javascript.**
   In JavaScript, output and input methods are used to interact with the user and display information to the user, as well as to collect data or input from the user. These methods are typically used in web development to create dynamic and interactive web applications.
   1. Output Methods: Output methods in JavaScript are used to display information to the user. They are mainly used to update the content of the web page or show messages in the browser.

   1) **console.log():** This method is used for debugging purposes and displays messages in the browser's developer console. It is not visible to the user on the web page itself.
      eg- console.log("This message will appear in the browser's console.");

2) **document.write():** This method writes HTML content directly to the web page. It can be used to dynamically add content to a web page, but it's generally not recommended for complex applications as it can overwrite the entire document if used after the page has loaded.
   eg- document.write("This text will be added to the web page.");
3) **innerHTML property**: You can use the `innerHTML` property of an HTML element to update its content. This is a common way to dynamically modify the content of elements on a web page.
   eg- document.getElementById("myElement").innerHTML = "New content here";
4) **alert():** As mentioned in the previous response, the `alert()` method displays a pop-up dialog box with a message, which is a way to alert the user to important information.
   eg- alert("This is an alert message.");
5) **confirm() -** A confirm box is often used if you want the user to verify or accept something. When a confirm box pops up, the user will have to click either "OK" or "Cancel" to proceed. If the user clicks "OK", the box returns true. If the user clicks "Cancel", the box returns false.
   Syntax- window.confirm("sometext");
   Eg-if (confirm("Press a button!")) {
   txt = "You pressed OK!";
   } else {
   txt = "You pressed Cancel!";
   }
6) **prompt() -** A prompt box is often used if you want the user to input a value before entering a page.When a prompt box pops up, the user will have to click either "OK" or "Cancel" to proceed after entering an input value. If the user clicks "OK" the box returns the input value. If the user clicks "Cancel" the box returns null.
   Syntax- window.prompt("sometext","defaultText");
   Eg- let person = prompt("Please enter your name", "Harry Potter");
   let text;
   if (person == null || person == "") {
   text = "User cancelled the prompt.";
   } else {
   text = "Hello " + person + "! How are you today?";
   }

2. Input Methods: Input methods in JavaScript are used to collect data or input from the user. They allow users to provide information or make choices within the web application.

1) **prompt():** As mentioned earlier, the `prompt()` method displays a pop-up dialog box with an input field, allowing the user to enter text.

eg- const userInput = prompt("Please enter your name:");

2) **HTML Form Elements:** In web development, HTML forms are used to collect various types of user input, such as text, checkboxes, radio buttons, and more. JavaScript is often used to access and process the data submitted through these forms.
eg-

```
<form>
<label for="name">Name:</label>
<input type="text" id="name" name="name">
 <input type="submit" value="Submit">
 </form>
 const form = document.querySelector("form");
 form.addEventListener("submit", function(event) {
 event.preventDefault(); // Prevent the form from submitting
const name = document.getElementById("name").value;
alert("Hello, " + name + "!");
  });
```

3) **Event Listeners:** JavaScript can listen for user events such as clicks, keystrokes, and mouse movements to collect input and trigger actions accordingly.
eg-

```
 const button = document.getElementById("myButton");
button.addEventListener("click", function() {
alert("Button clicked!");
});
```

3. **What are functions ? Explain how to create and call functions.**
A function is a set of statements that take inputs, do some specific computation, and produce output. The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can call that function.
We can create functions in JavaScript using the keyword function.
Syntax: The basic syntax to create a function in JavaScript is shown below:

```
function functionName(Parameter1, Parameter2, ...)
{
   // Function body
}
```

To create a function in JavaScript, we have to first use the keyword function, separated by the name of the function and parameters within parenthesis. The part of the function inside the curly braces {} is the body of the function. In javascript, functions can be used in the same way as variables for assignments, or calculations.
Function Definition: Before, using a user-defined function in JavaScript we have to create one. We can use the above syntax to create a function in JavaScript. Below are the rules for creating a function in JavaScript:

Every function should begin with the keyword function followed by,

- A user-defined function name that should be unique,
- A list of parameters enclosed within parentheses and separated by commas,
- A list of statements composing the body of the function enclosed within curly braces {}.

There are three ways of writing a function in JavaScript:

1) Function Declaration: It declares a function with a function keyword. The function declaration must have a function name.

Syntax-

```
function geeksforGeeks(paramA, paramB) {
   // Set of statements
}
```

2) Function Expression: It is similar to a function declaration without the function name. Function expressions can be stored in a variable assignment.

Syntax- let geeksforGeeks= function(paramA, paramB) {

```
   // Set of statements
}
```

3) Arrow Function: It is one of the most used and efficient methods to create a function in JavaScript because of its comparatively easy implementation. It is a simplified as well as a more compact version of a regular or normal function expression or syntax.

Syntax- let function_name = (argument1, argument2 ,..) => expression

Calling Functions: After defining a function, the next step is to call them to make use of the function. We can call a function by using the function name separated by the value of parameters enclosed between the parenthesis and a semicolon at the end. The below syntax shows how to call functions in JavaScript:

functionName(Value1, Value2, ..);

eg- function welcomeMsg(name) {

```
   return ("Hello " + name + " welcome to GeeksforGeeks");
}
// creating a variable
let nameVal = "Admin";
// calling the function
console.log(welcomeMsg(nameVal));
```

4. **Explain various datatypes used in Javascript.**

JavaScript is a dynamically-typed language, which means variables can hold values of different data types at different times during execution. JavaScript has several built-in data types that can be categorized into two main groups: primitive data types and non-primitive data types.

Primitive Data Types:

1.Number: Represents both integer and floating-point numbers. Examples:

```
let age = 25;
let price = 99.99;
```

2. String: Represents sequences of characters (text). Strings can be enclosed in single or double quotes. Example-

```
let name = "John";
let message = 'Hello, world!';
```

3. Boolean: Represents true or false values. Example-

```
let isTrue = true;
let isFalse = false;
```

4. Undefined: Represents a variable that has been declared but has not been assigned a value yet. Example-

```
let undefinedVar;
```

5. Null: Represents the intentional absence of any object value. Example-

```
let emptyValue = null;
```

6. Symbol(ES6): Represents a unique and immutable value, often used as object property keys. Example-

```
const uniqueSymbol = Symbol("description");
```

Non-primitive Data Types:

1. Object: Represents a collection of key-value pairs and is one of the most versatile data types in JavaScript. Objects can store various data types and functions. Example-

```
let person = {
  firstName: "John",
  lastName: "Doe",
  age: 30,
  hobbies: ["Reading", "Gardening"],
  sayHello: function() {
    console.log("Hello, world!");
  }
};
```

2. Array: Represents an ordered list of values. Arrays can contain elements of different data types. Example-

```
let colors = ["red", "green", "blue"];
```

3. Function: Functions are objects in JavaScript. They can be assigned to variables, passed as arguments, and returned from other functions. Example-

```
function add(a, b) {
  return a + b;
}
```

4. RegExp: Represents regular expressions for pattern matching and manipulation of strings. Example-

```
let regex = /pattern/;
```

5. Map(ES6): Represents a collection of key-value pairs with unique keys. It allows you to store and retrieve values based on their keys.

```
let myMap = new Map();
myMap.set("name", "John");
```

6. Set (ES6): Represents a collection of unique values. It is used to store and manage a list of distinct values. Example-

```
let mySet = new Set([1, 2, 3, 2, 1]);
```

5. **Different ways of declaring variables in Java Script(let,var,const).**

In JavaScript, you can declare variables using three different keywords: `let`, `var`, and `const`. Each of these keywords has its own behavior and use cases. Here's an explanation of each:

1. let: let was introduced in ECMAScript 6 (ES6) and is now the recommended way to declare variables in most situations.

- Variables declared with `let` are block-scoped, which means they are only accessible within the block (enclosed by curly braces) where they are defined.
- You can reassign values to variables declared with `let`.

Example:

```
let age = 25; // Declare and initialize a variable
if (true) {
  let name = "John"; // Block-scoped variable
  age = 30; // You can reassign the value
}
console.log(age); // 30
console.log(name); // ReferenceError: name is not defined
```

2. var: var has been available in JavaScript since its early versions. However, it has some quirks and limitations compared to `let` and `const`.

- Variables declared with `var` are function-scoped, meaning they are only accessible within the function where they are defined, or they have global scope if declared outside any function.
- Variables declared with `var` can be redeclared within the same scope, which can lead to unexpected behavior.

Example:

```
var age = 25; // Declare and initialize a variable
if (true) {
  var name = "John"; // Function-scoped variable
  age = 30; // You can reassign the value
}
console.log(age); // 30
console.log(name); // "John" - accessible outside the block
```

3. const: const is also introduced in ECMAScript 6 (ES6) and is used to declare variables whose values should not be reassigned after their initial assignment.

- Variables declared with `const` are block-scoped, just like `let`.
- You must assign a value to a `const` variable at the time of declaration, and you cannot reassign it later.

Example:

```
const age = 25; // Declare and initialize a constant
if (true) {
  const name = "John"; // Block-scoped constant
  age = 30; // Error: Assignment to constant variable
}
console.log(age); // 25
console.log(name); // ReferenceError: name is not defined
```

6. **What is an event handler? explain any three mouse events.**

An event handler in JavaScript is a function that is used to respond to events triggered by user interactions or other sources in a web page or application. Event handlers are responsible for defining what should happen when a specific event occurs, such as a user clicking a button, moving the mouse, typing on the keyboard, or any other action that generates an event. Event handlers are an essential part of building interactive and dynamic web applications.

Here are three common mouse events in JavaScript, along with explanations:

1. click Event:

- The `click` event is triggered when the user clicks the mouse button (usually the left button) on an element, like a button, link, or any other clickable element.
- You can attach a `click` event handler to an element to perform an action when it is clicked by the user.

Example:

```
const button = document.getElementById("myButton");
button.addEventListener("click", function() {
  console.log("Button clicked!");
});
```

2. mouseover Event:

- The `mouseover` event is triggered when the mouse pointer enters the boundaries of an element.
- It is commonly used to provide feedback or make elements change when the mouse hovers over them.

Example:

```
const element = document.getElementById("myElement");
element.addEventListener("mouseover", function() {
  element.style.backgroundColor = "lightblue";
});
```

3. mouseout Event:

- The `mouseout` event is triggered when the mouse pointer leaves the boundaries of an element, moving away from it.

- It is often used to revert changes made during a `mouseover` event, returning the element to its original state.

Example:

```
const element = document.getElementById("myElement");
element.addEventListener("mouseover", function() {
  element.style.backgroundColor = "lightblue";
});
element.addEventListener("mouseout", function() {
  element.style.backgroundColor = ""; // Revert to the default background color
});
```

These are just a few examples of mouse events in JavaScript. There are many more mouse-related events, such as `mousedown`, `mouseup`, `mousemove`, and `contextmenu`, which can be used to build rich and interactive user interfaces.

7. **Different types of events and event handlers.**

Events in JavaScript represent a signal that something has happened. This could be a user action (like a button click, form submission, or key press), or it could be a browser action (like a page load). JavaScript allows you to react to these events and execute code when they occur. There are many different types of events that can occur. Some common ones include:

click: User clicks on an HTML element
dblclick: User double-clicks on an HTML element
mousedown, mouseup, mousemove: User performs a mouse action
keydown, keyup: User presses or releases a keyboard key
load: A page or an image is finished loading
blur, focus: An element loses or gets focus
scroll: User scrolls a webpage
resize: The browser window is resized

Event handlers, also known as event listeners, are the functions that are invoked when an event occurs.Event handlers can be assigned to HTML elements to react to the events that occur on those elements. The most flexible way to assign an event handler is to use the addEventListener method, which allows multiple listeners to be assigned to an element, and for listeners to be removed if needed.

Example of adding a click event listener to a button:

```
const btn = document.querySelector("button");
function greet(event) {
  console.log("greet:", event);
}
btn.addEventListener("click", greet);
```

In addition to addEventListener, you can also assign event handlers directly to the on event properties of HTML elements. For example:

```
const btn = document.querySelector("button");
btn.onclick = () => {
  console.log("Button clicked");
};
```

Finally, it's important to note that you can remove event listeners using the removeEventListener method, which is useful for cleaning up event handlers that are no longer needed:

```
btn.removeEventListener("click", greet);
```

8. **What do you mean by an object in java script different ways of creating objects in Java Script ?**

In JavaScript, an object is a fundamental data structure that represents a collection of key-value pairs. Objects are used to store and organize data, and they can contain various data types, including other objects, functions, and primitive values like numbers and strings. Objects are at the core of JavaScript programming and play a central role in structuring and modeling data.

There are several ways to create objects in JavaScript:

1. Object Literal:

- The simplest way to create an object is by using an object literal notation, which involves enclosing key-value pairs in curly braces `{}`.

Example-

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 30,
};
```

2. Constructor Function:

- You can create objects using constructor functions. These functions are like templates for creating multiple objects with similar properties and methods.

Example-

```
function Person(firstName, lastName, age) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.age = age;
}
const person = new Person("John", "Doe", 30);
```

3. ES6 Class:

- ES6 introduced a more concise way to create constructor functions using classes.

Example-

```
class Person {
  constructor(firstName, lastName, age) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
  }
}
const person = new Person("John", "Doe", 30);
```

4. Object.create():

- The `Object.create()` method allows you to create a new object with a specified prototype object.

Example-

```
const personPrototype = {
  greet: function() {
    console.log(`Hello, my name is ${this.firstName} ${this.lastName}.`);
  },
};
const person = Object.create(personPrototype);
person.firstName = "John";
person.lastName = "Doe";
```

5. Using Spread Operator (ES6):

- You can create new objects by spreading the properties of an existing object into a new one.

Example-

```
const person = {
  firstName: "John",
  lastName: "Doe",
};
const newPerson = { ...person, age: 30 };
```

9. **Explain Object Oriented concepts in JavaScript.**

JavaScript is a versatile and object-oriented programming language that supports a range of object-oriented concepts and paradigms. Understanding these concepts is essential for effective JavaScript development. Here are some key object-oriented concepts in JavaScript:

1. Objects:

- In JavaScript, objects are the most fundamental construct. They are collections of key-value pairs, where keys (properties) are strings, and values can be of any data type, including other objects, functions, and primitive types.
- Objects can be created using object literal notation, constructor functions, ES6 classes, or `Object.create()`.

2. Constructor Functions and Classes:

- JavaScript allows you to create constructor functions and ES6 classes to define blueprints for creating objects. These constructors/classes can have properties and methods.
- Instances of objects can be created using the `new` keyword followed by the constructor function or class name.

Example-

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}
const person = new Person("John", "Doe");
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
}
const person = new Person("John", "Doe");
```

3. Inheritance:

- JavaScript supports prototypal inheritance, where objects can inherit properties and methods from other objects (prototypes). This is achieved through the `prototype` property.
- ES6 classes provide a more structured way to define inheritance using the `extends` keyword.

Example-

```
function Animal(name) {
  this.name = name;
}
Animal.prototype.eat = function() {
  console.log(`${this.name} is eating.`);
};
function Dog(name, breed) {
  Animal.call(this, name);
  this.breed = breed;
}
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;
const myDog = new Dog("Buddy", "Golden Retriever");
myDog.eat(); // "Buddy is eating."

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

```
  eat() {
    console.log(`${this.name} is eating.`);
  }
}

class Dog extends Animal {
  constructor(name, breed) {
    super(name);
    this.breed = breed;
  }
}
const myDog = new Dog("Buddy", "Golden Retriever");
myDog.eat(); // "Buddy is eating."
```

4. Encapsulation and Information Hiding:

- JavaScript doesn't have native access modifiers like `private` or `protected`, but you can achieve encapsulation and information hiding through closures and naming conventions.
- By encapsulating data within objects and using getter and setter methods, you can control access to object properties.

Example-

```
function Counter() {
  let count = 0;
  this.increment = function() {
    count++;
  };
  this.getCount = function() {
    return count;
  };
}
const counter = new Counter();
counter.increment();
console.log(counter.getCount()); // 1
```

10. Explain advantages of using Arrow function.

Arrow functions, introduced in ECMAScript 6 (ES6), are a concise way to write anonymous functions in JavaScript. They provide a more compact and often more readable syntax compared to traditional function expressions.

Advantages of Arrow Functions

- Arrow functions reduce the size of the code.
- The return statement and function brackets are optional for single-line functions.
- Provides the feature of creating anonymous functions.
- It increases the readability of the code.

- Arrow functions provide a lexical ‘this’ binding. It means, they inherit the value of “this” from the enclosing scope. This feature can be advantageous when dealing with event listeners or callback functions where the value of “this” can be uncertain.

Disadvantages of Arrow Functions

- No Binding of this: While the lexical this behavior is an advantage in many cases, it can be a disadvantage when you need to access the function's own this. Arrow functions are not suitable for methods that require their own ‘this’ context.
- Cannot Be Used as Constructors: Arrow functions cannot be used with the new keyword to create instances. They lack the necessary internal behavior to support object construction.
- Limited Functionality: Due to their concise syntax and lack of a block body, arrow functions are best suited for simple, one-liner expressions. For more complex functions that require multiple statements or complex logic, traditional function expressions are often more appropriate.
- Readability in Complex Cases: In complex code scenarios, arrow functions might lead to less readable code if used excessively, especially when it becomes unclear which surrounding context the function is inheriting.

11. Explain generator function and how it is different from normal function.

ES6 introduces a new concept called Generator (or Generator function). It gives you a new way to work with iterators and functions. The ES6 generator is a new type of function that can be paused in the middle or many times in the middle and resumed later. In the standard function, control remains with the called function until it returns, but the generator function in ES6 allows the caller function to control the execution of a called function.

The difference between a generator and a regular function is:

- In response to a generator call, its code doesn’t run. In its place, it returns a special object called a ‘Generator Object’ to manage the execution.
- At any time, the generator function can return (or yield) the control back to the caller.
- The generator can return (or yield) multiple values according to the requirement, unlike a regular function.
- Syntax: Generator functions have a similar syntax to regular functions. As the only difference, the generator function is denoted by an asterisk (*) after the function keyword.

Ex: function *myfunction() { }

Yield operator: In the yield statement, function execution is suspended and a value is returned to the caller. In this case, enough state is retained for the function to resume where it left off. Following the last yield run, the function resumes execution immediately after it has been resumed. You can produce a series of values using this function.

12. Synchronous and Asynchronous Programming

S.No	Synchronous JavaScript	Asynchronous JavaScript
1.	Every statement in a function is executed sequentially.	Every statement in a function is not executed sequentially. It can also execute in parallel.
2.	The execution of the next instruction depends upon the execution of the previous instruction.	The next instruction can execute while the previous one is still executing.
3.	Synchronous JavaScript is preferred more.	It is preferred in situations when the execution of some process gets blocked indefinitely.
4.	Synchronous is a single thread.	Asynchronous is multi-thread in nature.
5.	Synchronous is slower in nature.	Asynchronous increases throughout as processes run in parallel.

13. Template Literal, Rest and Spread,default Argument

Ans

Template Literals: Template literals are literals delimited with backtick (`) characters, allowing for multi-line strings, string interpolation with embedded expressions, and special constructs called tagged templates.

Template literals are sometimes informally called template strings, because they are used most commonly for string interpolation (to create strings by doing substitution of placeholders). However, a tagged template literal may not result in a string; it can be used with a custom tag function to perform whatever operations you want on the different parts of the template literal.

Spread: Spread operator: The spread operator(denoted by …) helps us expand an iterable such as an array where multiple arguments are needed, it also helps to expand the object expressions. In cases where we require all the elements of an iterable or object to help us achieve a task, we use a spread operator.

Example:

Code:

```
var array1 = [10, 20, 30, 40, 50];
 var array2 = [60, 70, 80, 90, 100];
 var array3 = [...array1, ...array2];
 console.log(array3);
```

O/p:
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

Rest: Rest operator: The rest parameter is converse to the spread operator. while the spread operator expands elements of an iterable, the rest operator compresses them. It collects several elements. In functions when we require to pass arguments but were not sure how many we have to pass, the rest parameter makes it easier.

Example:
Code:

```
function average(...args) {
      console.log(args);
      var avg =
         args.reduce(function (a, b) {
            return a + b;
        }, 0) / args.length;
      return avg;
  }
  console.log("average of numbers is : "
     + average(1, 2, 3, 4, 5));
  console.log("average of numbers is : "
     + average(1, 2, 3));
```

O/p:
[1, 2, 3, 4, 5]
"average of numbers is : 3"
[1, 2, 3]
"average of numbers is : 2"

Default argument: Default parameters are the default values for the arguments whose values are not passed during the function call.Default function parameters allow named parameters to be initialized with default values if no value or undefined is passed.

Example:

```
function multiply(a, b = 1) {
  return a * b;
}

console.log(multiply(5, undefined));
//  output: 5

console.log(multiply(5));
//  output: 5
```

14. **Differentiate:**
    a. **Function and Arrow function**
    b. **Generator and Iterator**
    c. **Let,Var,Const**

Ans:

a.

Aspect	Regular Function	Arrow Function
Syntax	`function functionName(parameters) { body }`	`(parameters) => expression`
`this` **Binding**	Own `this` context determined at runtime	Inherited `this` from surrounding lexical context
Arguments Object	Has its own `arguments` object	Does not have its own `arguments` object
Use Cases	Versatile; can be used in various scenarios	Best suited for simple, one-liner expressions
Block Bodies	Can have block bodies with multiple statements	Can have a single expression without braces
Constructor Usage	Can be used with `new` for object instantiation	Cannot be used as constructors
Method Usage	Can be used as object methods	Can be used as object methods, but with inherited `this`
Generator Functions	Can be used with generator functions (e.g., `function*`)	Cannot be used with generator functions

1. Syntax: Regular function.

```
let square = function(x){
    return (x*x);
};
console.log(square(9));
```

Syntax: Arrow function.

```
var square = (x) => {
    return (x*x);
};
console.log(square(9));
```

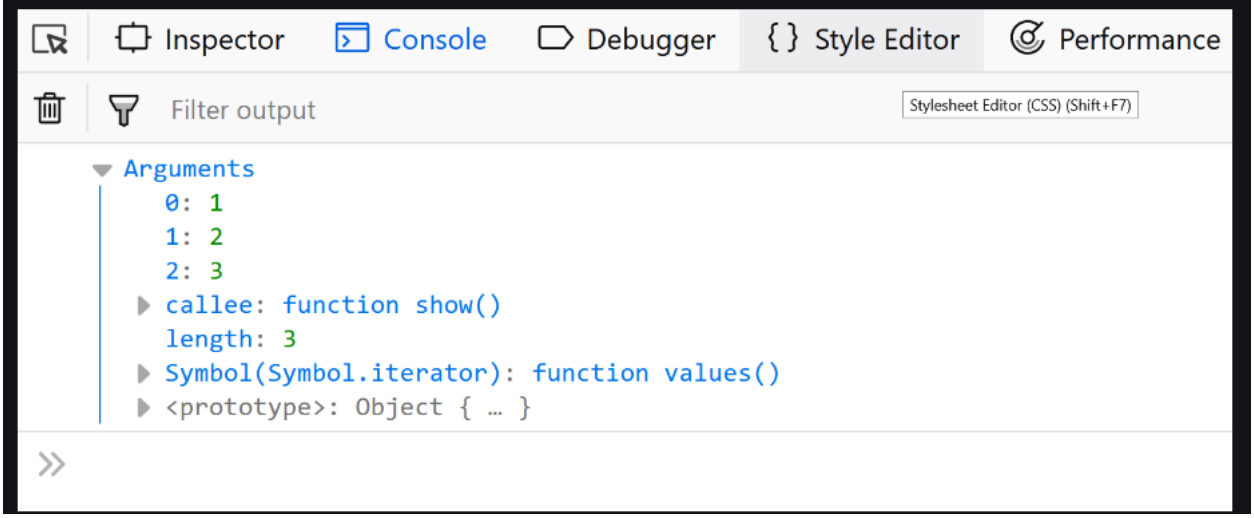
2. Use of this keyword: Unlike regular functions, arrow functions do not have their own this.

3. Availability of arguments objects: Arguments objects are not available in arrow functions, but are available in regular functions. Example using regular ():

javascript

```
let user = {
    show(){
        console.log(arguments);
    }
};
user.show(1, 2, 3);
```

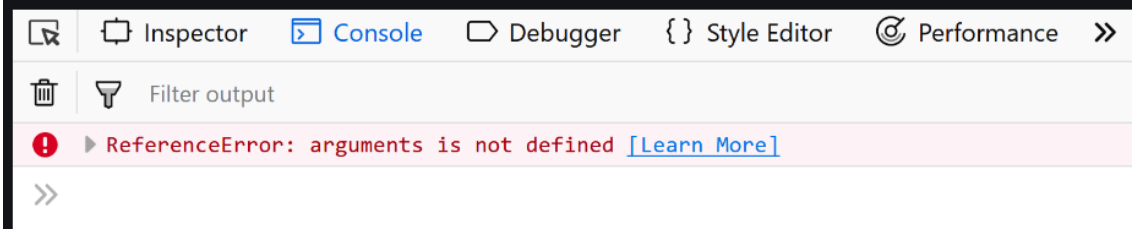
Output:



javascript

```
let user = {
        show_ar : () => {
        console.log(...arguments);
    }
};
user.show_ar(1, 2, 3);
```

Output:



b.

Aspect	Iterators	Generators
Definition	Objects with a `next()` method.	Functions created using `function*` syntax.
Usage	Explicit iteration, `next()` method.	Automatic pausing and resuming with `yield`.
State Management	External, manual state management.	Internal state management.
Infinite Sequences	Possible but requires careful handling.	Well-suited for generating infinite sequences.
Syntax	No specific syntax, implemented manually.	Created using `function*` and `yield` statements.
Next Value Retrieval	Values retrieved using `next()` method.	Values generated using `yield`.
Pausing and Resuming	No built-in pausing and resuming.	Automatic pausing and resuming.
Error Handling	Manual error handling within the iterator.	Errors can propagate from generator functions.
Memory Efficiency	More memory-efficient for finite sequences.	Memory usage can be higher for long-running generators.
Use Cases	Suitable for simple iteration over collections.	Used for complex or infinite sequences, asynchronous operations, and lazy evaluation.

c.

var	let	const
The scope of a *var* variable is functional scope.	The scope of a *let* variable is block scope.	The scope of a *const* variable is block scope.
It can be updated and re-declared into the scope.	It can be updated but cannot be re-declared into the scope.	It cannot be updated or re-declared into the scope.
It can be declared without initialization.	It can be declared without initialization.	It cannot be declared without initialization.
It can be accessed without initialization as its default value is “undefined”.	It cannot be accessed without initialization otherwise it will give ‘referenceError’.	It cannot be accessed without initialization, as it cannot be declared without initialization.
hoisting done, with initializing as ‘default’ value	Hoisting is done, but not initialized (this is the reason for the error when we access the let variable before declaration/initialization	Hoisting is done, but not initialized (this is the reason for the error when we access the const variable before declaration/initialization

15. **Short note on following:**
    a. **Promise**
    b. **Callback function**
    c. **Generator and Iterator**

Answer:

a. JavaScript Promise are easy to manage when dealing with multiple asynchronous operations where callbacks can create callback hell leading to unmanageable code. Prior to promises events and callback functions were used

but they had limited functionalities and created unmanageable code. Multiple callback functions would create callback hell that leads to unmanageable code. Promises are used to handle asynchronous operations in JavaScript.

Javascript

```
let promise = new Promise(function (resolve, reject) {
    const x = "geeksforgeeks";
    const y = "geeksforgeeks"
    if (x === y) {
        resolve();
    } else {
        reject();
    }
});

promise.
    then(function () {
        console.log('Success, You are a GEEK');
    }).
    catch(function () {
        console.log('Some error has occurred');
    });
```

Output:

```
Success, You are a GEEK
```

b. A JavaScript callback is a function which is to be executed after another function has finished execution.

A more formal definition would be - Any function that is passed as an argument to another function so that it can be executed in that other function is called as a callback function.

We need callback functions because many JavaScript actions are asynchronous, which means they don't really stop the program (or a function) from running until they're completed, as you're probably used to. Instead, it will execute in the background while the rest of the code runs.

c. Same as prev Q

16. Java Script DOM and different methods of Document object

JavaScript Document Object Model (DOM) is a programming interface that represents and interacts with the structure and content of web documents. It allows you to manipulate HTML and XML documents, making it possible to dynamically update, change, and interact with web pages. The central object in the DOM is the Document object, which represents the web page itself. Here are some key methods and properties of the Document object:

Common Properties and Methods of the Document Object:

Method/Property	Description
`getElementById(id)`	Get element by its `id` attribute.
`getElementsByClassName(className)`	Get elements by class name.
`getElementsByTagName(tagName)`	Get elements by HTML tag name.
`querySelector(selector)`	Get the first element matching a CSS selector.
`querySelectorAll(selector)`	Get all elements matching a CSS selector.
`createElement(tagName)`	Create a new HTML element.
`appendChild(node)`	Add a node as the last child of another node.
`removeChild(node)`	Remove a node from the DOM.
`replaceChild(newNode, oldNode)`	Replace an old node with a new node.
`addEventListener(event, function)`	Attach an event listener to the document.

These are just a few of the many methods and properties available on the Document object. The DOM provides a powerful way to interact with and manipulate web content, allowing you to build dynamic and interactive web applications.

17. **Data validation in java Script and Regular expression.**

Data validation in JavaScript often involves the use of regular expressions (regex) to check and ensure that user-provided data conforms to specific patterns or formats. Regular expressions are powerful tools for pattern matching and are commonly used for validating data such as email addresses, phone numbers, passwords, and more. Here's an overview of data validation in JavaScript using regular expressions:

1. Regular Expressions (Regex):

- A regular expression is a pattern that describes a set of strings. It is a powerful tool for text pattern matching and manipulation.
- In JavaScript, you can create regex patterns using literal notation (enclosed in slashes `/`) or the `RegExp` constructor.

Eg-

```
// Literal notation
const emailRegex = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$/;
```

```
// RegExp constructor
const phonePattern = new RegExp(/^\d{10}$/);
```

2. Data Validation:
- You can use regular expressions to validate various types of data, such as email addresses, phone numbers, dates, and more.

Eg-

```
const email = "example@email.com";
if (emailRegex.test(email)) {
  console.log("Email is valid.");
} else {
  console.log("Email is invalid.");
}
```

3. Common Validation Patterns:
- Here are some common regular expression patterns for data validation:
  - Email address validation: `/^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$/`
  - Phone number validation (10 digits): `/^\d{10}$/`
  - Date format validation (YYYY-MM-DD): `/^\d{4}-\d{2}-\d{2}$/`
  - Password validation (at least 8 characters, containing letters, numbers, and special characters): `^(?=.*[A-Za-z])(?=.*\d)(?=.*[@$!%*#?&])[A-Za-z\d@$!%*#?&]{8,}$`

4. Regex Methods:
- JavaScript provides several regex methods for data validation:
  - `test()`: Tests a string against a regex pattern and returns `true` or `false` based on whether the string matches the pattern.
  - `match()`: Searches a string for a pattern match and returns an array of matching substrings.
  - `search()`: Searches a string for a pattern match and returns the index of the first match.
  - `replace()`: Replaces matches of a pattern with a specified replacement.

Eg-

```
const text = "Sample text with a phone number: 1234567890";
const phonePattern = /\d{10}/;
const isValidPhone = phonePattern.test(text); // Returns true
```

5. Error Handling:
- When data validation fails, you should provide meaningful error messages to the user to explain why their input is invalid.

Data validation using regular expressions is a crucial aspect of ensuring data integrity and security in web applications. It helps prevent invalid or malicious data from being processed, enhancing the overall quality of user interactions.

Sample Programs:

1. **Write a javascript code to find sum of N natural numbers using user define function.**

```
// Function to find the sum of N natural numbers
function sumOfNaturalNumbers(N) {
  let sum = 0;
  for (let i = 1; i <= N; i++) {
    sum += i;
  }
  return sum;
}
// Get input from the user
const N = parseInt(prompt("Enter a positive integer (N):"));
// Check if N is a positive integer
if (Number.isInteger(N) && N > 0) {
  const result = sumOfNaturalNumbers(N);
  console.log(`The sum of the first ${N} natural numbers is: ${result}`);
} else {
  console.log("Please enter a positive integer.");
}
```

2. **Write a javascript code to find sum of N natural numbers using user defined function. SAME AS 1st code**

3. **Write Javascript code that will change the background color of the page when user clicks on the particular button.**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Change Background Color</title>
    <style>
        body {
            transition: background-color 0.5s;
        }
    </style>
</head>
<body>
    <button id="changeColorButton">Change Color</button>
    <script>
        // Function to change the background color
        function changeBackgroundColor() {
```

```
        const colors = ["#FF5733", "#33FF57", "#5733FF", "#FFFF33"]; // Add your
desired colors here
        const randomColor = colors[Math.floor(Math.random() * colors.length)];
        document.body.style.backgroundColor = randomColor;
      }

      // Add a click event listener to the button
      const changeColorButton =
document.getElementById("changeColorButton");
      changeColorButton.addEventListener("click", changeBackgroundColor);
    </script>
</body>
</html>
```

4. **Write Javascript code to calculate gross salary whare salary details has been accepted using form.**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Gross Salary Calculator</title>
</head>
<body>
    <h1>Gross Salary Calculator</h1>
    <form id="salaryForm">
        <label for="basicSalary">Basic Salary:</label>
        <input type="number" id="basicSalary" required><br>

        <label for="allowances">Allowances:</label>
        <input type="number" id="allowances" required><br>

        <label for="deductions">Deductions:</label>
        <input type="number" id="deductions" required><br>

        <input type="button" value="Calculate Gross Salary"
onclick="calculateGrossSalary()">
    </form>

    <h2>Result:</h2>
    <p id="result">Gross Salary: <span id="grossSalary">-</span></p>

    <script>
        function calculateGrossSalary() {
```

```
        const basicSalary =
parseFloat(document.getElementById("basicSalary").value);
        const allowances =
parseFloat(document.getElementById("allowances").value);
        const deductions =
parseFloat(document.getElementById("deductions").value);

        if (!isNaN(basicSalary) && !isNaN(allowances) && !isNaN(deductions)) {
          const grossSalary = basicSalary + allowances - deductions;
          document.getElementById("grossSalary").textContent =
`$${grossSalary.toFixed(2)}`;
        } else {
          document.getElementById("grossSalary").textContent = "Invalid input";
        }
      }
    </script>
</body>
</html>
```

React Sample Questions:

1. **What is react? Explain the Features of React.**

   React is a popular open-source JavaScript library for building user interfaces (UIs). It was developed and is maintained by Facebook and a community of individual developers and companies. React is widely used for creating interactive, dynamic, and efficient web applications. Here are some key features of React:

   1. Component-Based Architecture:
      - React follows a component-based architecture, where UIs are divided into reusable, self-contained components. Components are like building blocks that can be composed to create complex user interfaces.
      - Component reusability simplifies development, maintenance, and testing.
   2. Virtual DOM (Document Object Model):
      - React uses a virtual DOM to optimize rendering performance. Instead of directly manipulating the browser's DOM, React creates a lightweight virtual representation of it.
      - When data changes, React updates the virtual DOM and efficiently calculates the minimal number of changes required to update the actual DOM, minimizing browser reflows and improving performance.
   3. Declarative Syntax:
      - React uses a declarative approach to describe how the UI should look based on the application's current state. Developers define the desired UI state, and React takes care of updating the actual DOM to match that state.
      - This makes it easier to reason about UI behavior and reduces the risk of bugs related to manual DOM manipulation.
   4. JSX (JavaScript XML):

- JSX is a syntax extension for JavaScript that allows you to write HTML-like code within JavaScript. React components are often written using JSX, making it easier to create and visualize the UI structure.
- JSX is transpiled to JavaScript for execution in the browser.

5. Unidirectional Data Flow:
- React enforces a unidirectional data flow, which means that data flows in one direction from parent components to child components. This makes it easier to understand how data changes affect the UI.
- State management in React can be organized using component state or external state management libraries like Redux or MobX.

6. Component Lifecycle Methods:
- React provides a set of lifecycle methods that allow developers to hook into the different stages of a component's lifecycle. These methods can be used for tasks like initialization, data fetching, and cleanup.
- Lifecycle methods help manage component behavior and side effects.

7. Rich Ecosystem:
- React has a vast ecosystem of libraries, tools, and extensions that enhance its functionality and make development more efficient. This includes React Router for routing, Redux for state management, and many UI component libraries.

8. Community Support:
- React has a large and active community of developers and organizations. This community contributes to the library's development, shares knowledge through documentation and tutorials, and provides support through forums and social media.

9. Server-Side Rendering (SSR):
- React supports server-side rendering, allowing you to render React components on the server and send pre-rendered HTML to the client. This can improve initial page load times and SEO.

10. Mobile Development:
- React can be used for mobile app development through React Native, a framework that enables the creation of native mobile applications for iOS and Android using React components.

2. **What is JSX? Explain how JSX is processed by Browser.**

JSX (JavaScript Syntax Extension or JavaScript XML) is an extension to JavaScript that provides an easier way to create UI components in React[1]. It allows developers to write HTML-like code inside a JavaScript file and is widely used in React development[7]. JSX looks similar to HTML in syntax, but it is actually a syntax extension of JavaScript[11]. Here's how JSX is processed by the browser:

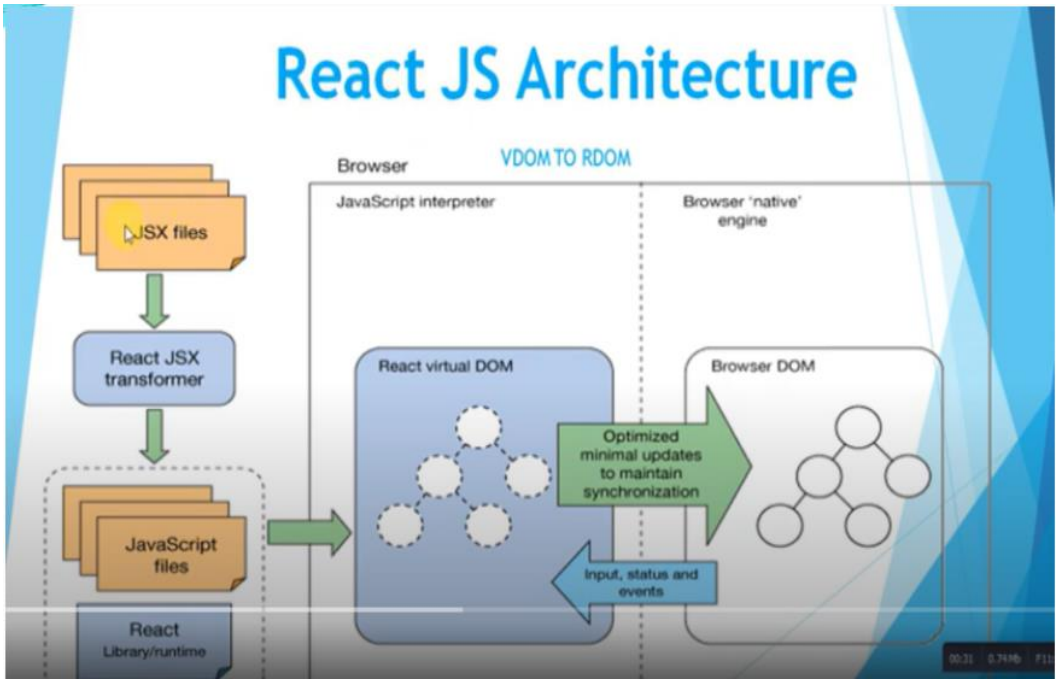
1) JSX code is transformed into regular JavaScript before it is executed in the browser. This transformation is done using a tool called a transpiler, with Babel being the most popular transpiler for JSX.

2) Babel transforms the JSX code into a series of function calls that are equivalent to the HTML-like code written in JSX. The browser can then execute the resulting JavaScript code.
3) The transformed JSX code can be written in a syntax that is familiar and easy to read, while still taking advantage of the power and flexibility of JavaScript.
4) The resulting JavaScript code is executed by the browser, rendering the UI components defined in the JSX code.

3. **What is Virtual DOM? How is it different from Real DOM?**
   1) VDOM is the virtual representation of Real DOM
   React update the state changes in Virtual DOM first and then it syncs with Real DOM
   2) Virtual DOM is just like a blueprint of a machine, can do changes in the blueprint but those changes will not directly apply to the machine.
   3) Virtual DOM is a programming concept where a virtual representation of a UI is kept in memory synced with “Real DOM ” by a library such as ReactDOM and this process is called reconciliation
   4) Virtual DOM makes the performance faster, not because the processing itself is done in less time. The reason is the amount of changed information – rather than wasting time on updating the entire page, you can dissect it into small elements and interactions

Real DOM	Virtual DOM
DOM manipulation is very expensive	DOM manipulation is very easy
There is too much memory wastage	No memory wastage
It updates Slow	It updates fast
It can directly update HTML	It can’t update HTML directly
Creates a new DOM if the element updates.	Update the JSX if the element update
It allows us to directly target any specific node (HTML element)	It can produce about 200,000 Virtual DOM Nodes / Second.
It represents the Ul of your application	It is only a virtual representation of the DOM

4. **The architecture of React.**



The React library is built on a solid foundation. It is simple, flexible and extensible. The primary objective of React is to enable the developer to create user interfaces using pure JavaScript.

- Typically, each user interface library introduces a new template language for designing user interfaces and provides the option to write logic inside the template or separately.
- Instead of introducing a new template language, React introduces three simple concepts as below -
- React elements- JavaScript representation of HTML DOM. React provides an API, React.createElement to create React Element.
- JSX- A JavaScript extension for designing user interfaces. JSX is an XML based, extensible language that supports HTML syntax with slight modifications. JSX can be compiled for React Elements and used to build user interfaces.
- React Component- React components are the primary building block of React applications. It uses React Elements and JSX to design its user interface.
- React Component is a JavaScript class (extends the React Component class) or a pure JavaScript function. React components have properties, state management, lifecycle and event handlers. React components can be able to do simple as well as advanced logic.

5. **Folder Structure of React application.**
   Folder Structure for React Project:
   1) Assets Folder
   2) Layouts Folder
   3) Components Folder
   4) Pages Folder

5) Middleware Folder
6) Routes Folder
7) Config Folder
8) Services Folder
9) Utils Folder

- **Assets Folder**

As the name says, it contains assets of our project. It consists of images and styling files. Here we can store our global styles. We are centralizing the project so we can store the page-based or component-based styles over here. But we can even keep style according to the pages folder or component folder also. But that depends on developer comfortability.

- **Layouts Folder**

As the name says, it contains layouts available to the whole project like header, footer, etc. We can store the header, footer, or sidebar code here and call it.

- **Components Folder**

Components are the building blocks of any react project. This folder consists of a collection of UI components like buttons, modals, inputs, loader, etc., that can be used across various files in the project. Each component should consist of a test file to do a unit test as it will be widely used in the project.

- **Pages Folder**

The files in the pages folder indicate the route of the react application. Each file in this folder contains its route. A page can contain its subfolder. Each page has its state and is usually used to call an async operation. It usually consists of various components grouped.

- **Middleware Folder**

This folder consists of middleware that allows for side effects in the application. It is used when we are using redux with it. Here we keep all our custom middleware.

- **Routes Folder**

This folder consists of all routes of the application. It consists of private, protected, and all types of routes. Here we can even call our sub-route.

- **Config Folder**

This folder consists of a configuration file where we store environment variables in config.js. We will use this file to set up multi-environment configurations in your application.

- **Services Folder**

This folder will be added if we use redux in your project. Inside it, there are 3 folders named actions, reducers, and constant subfolders to manage states. The actions and reducers will be called in almost all the pages, so create actions, reducers & constants according to pages name.

- **Utils Folder**

Utils folder consists of some repeatedly used functions that are commonly used in the project. It should contain only common js functions & objects

like dropdown options, regex condition, data formatting, etc.

6. **How do you create a React app?**
   To create a React app, you can use a tool called "Create React App" (CRA), which is an officially supported way to set up a new React project with a predefined folder structure, build configuration, and development environment. Here are the steps to create a React app using Create React App:

   Prerequisites:
   1. Ensure you have Node.js and npm (Node Package Manager) installed on your computer. You can download and install them from the official Node.js website if you haven't already: https://nodejs.org/

   Steps:
   1. Install Create React App globally:
      Open your terminal or command prompt and run the following command to install Create React App globally on your system: **npm install -g create-react-app**
   2. Create a new React app:
      After installing Create React App, you can create a new React app by running the following command, replacing `my-react-app` with your preferred project name: **npx create-react-app my-react-app**
      This command will set up a new React project in a folder called`my-react-app`.
   3. Navigate to the project folder:
      Change your current directory to the newly created project folder:
      cd my-react-app
   4. Start the development server:
      To start the development server and see your React app in action, run the following command: **npm start**
      This command will start the development server, and your React app will be available at `http://localhost:3000` in your web browser.
   5. Edit your React app:
      Your React app's source code is located in the `src` folder within the project directory. You can open this folder in your code editor and start editing the `src` files to build your app. The main entry point is usually `src/index.js`.
   6. Additional Setup (Optional):
      Depending on your project requirements, you may want to add additional libraries or components to your app. You can do this using npm or yarn. For example:
      - To install additional packages: Use `npm install` or `yarn add` followed by the package name.
      - To create new React components: You can create new component files in the `src` folder.
   7. Build and Deploy (Optional):

When you're ready to deploy your React app to a production environment, you can create a production-ready build by running: **npm run build**

This will create an optimized build of your app in the `build` folder, which you can then deploy to a web server or a hosting service of your choice.

7. **What are the components in React? Explain different types of Components?**

A Component is one of the core building blocks of React. In other words, we can say that every application you will develop in React will be made up of pieces called components. Components make the task of building UIs much easier. You can see a UI broken down into multiple individual pieces called components and work on them independently and merge them all in a parent component which will be your final UI.

In React, we mainly have two types of components:

1) **Functional Components:** Functional components are simply javascript functions. We can create a functional component in React by writing a javascript function. These functions may or may not receive data as parameters, we will discuss this later in the tutorial. The below example shows a valid functional component in React:

```
function demoComponent() {
   return (<h1>
            Welcome Message!
          </h1>);
}
```

2) **Class Components:** The class components are a little more complex than the functional components. The functional components are not aware of the other components in your program whereas the class components can work with each other. We can pass data from one class component to another class component. We can use JavaScript ES6 classes to create class-based components in React. The below example shows a valid class-based component in React:

```
class Democomponent extends React.Component {
   render() {
      return <h1>Welcome Message!</h1>;
   }
}
```

Rendering Components in ReactJS

React is also capable of rendering user-defined components. To render a component in React we can initialize an element with a user-defined component and pass this element as the first parameter to ReactDOM.render() or directly pass the component as the first argument to the ReactDOM.render() method. The below syntax shows how to initialize a component to an element:

```
const elementName = <ComponentName />;
```

8. **What is the use of render() in React?**

React renders HTML to the web page by using a function called render().

The purpose of the function is to display the specified HTML code inside the specified HTML element.
In React, the `render()` method is a fundamental and required method for class components. It plays a crucial role in the component's lifecycle, and its primary purpose is to define what should be displayed on the user interface.

Here's the main use of the `render()` method in React:
Defining the UI: The primary purpose of the `render()` method is to return a JSX (JavaScript XML) description of what the component should render. This JSX defines the structure and content of the component's output on the web page.In the render() method, we can read props and state and return our JSX code to the root component of our app.

```
class MyComponent extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, React!</h1>
        <p>This is a React component.</p>
      </div>
    );
  }
}
```

In the example above, the `render()` method defines the structure of the component, which consists of an `<h1>` element and a `<p>` element. This JSX will be translated into HTML and displayed in the user's web browser.

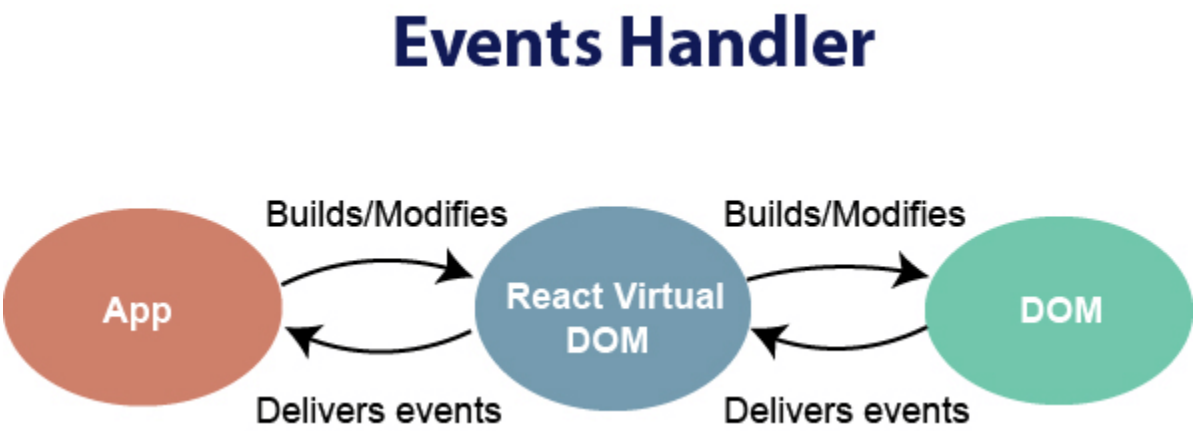
Key points to remember about the `render()` method:
1. The `render()` method must return a single JSX element or a fragment (introduced in React 16.2) encapsulating multiple elements.
2. The `render()` method should be a pure function, meaning it should not have any side effects, perform asynchronous tasks, or modify component state directly. Its purpose is solely to describe the UI based on the current state and props.
3. Whenever the component's state or props change, React will re-invokes the `render()` method to update the UI to reflect the new data.
4. The `render()` method should not modify the DOM directly. React abstracts the DOM manipulation through a virtual representation (Virtual DOM) and takes care of updating the actual DOM efficiently.

9. **What is an event in React? Explain how Events are handled in React.**
An event is an action that could be triggered as a result of the user action or system generated event. For example, a mouse click, loading of a web page, pressing a key, window resizes, and other interactions are called events.

camelCase Convention: Instead of using lowercase for naming, we use camelCase while giving names of the react events. That simply means we write ‘onClick’ instead of ‘onclick’.

Pass the event as a function: In React, we pass a function enclosed by curly brackets as the event listener or event handler, unlike HTML, where we pass the event handler or event listener as a string.

Prevent the default: Unlike javascript, returning false inside the JSX element does not prevent the default behavior in react. Instead, we have to call the ‘preventDefault’ method directly inside the event handler function



From the developer side, we need to ‘listen’ to such events and then make our application respond accordingly. This is called event handling that provides a dynamic interface to a webpage. Like JavaScript DOM, React also provides us with some built-in methods to create a listener that responds accordingly to a specific event. Javascript has events to provide a dynamic interface to a webpage. These events are hooked to elements in the Document Object Model(DOM). These events by default use bubbling propagation i.e, upwards in the DOM from children to parent. We can bind events either as inline or in an external script.

React Event Handling Example:

```
Import React from ‘…react’
function clickAppHandler() {
   function clickHandler() {
      console.log(‘clicked’)
   }
   return (
      <div>
         <button onClick={clickHandler}>Click</button>
      </div>
   )
}
```

```
export default clickAppHandler
```

As illustrated below, the clickHandler function is called when the onClick event occurs, and the console prints the word clicked when the button is clicked.

A component must then be added to the app component after this. You can see in the code above that when the button is clicked, the function is passed as the event handler. You will notice that we haven't added parentheses as it becomes a function, and we do not want that we want the handler to be a function not a function call. When a new component is rendered its event handler functions are added to the mapping maintained by the react. When the event is triggered and it hits and DOM object, react maps the event to the handler, if it matches it calls the handler

10. **What are synthetic events in React?(GPT)**

In React, synthetic events are a layer of abstraction over native browser events. React uses this system to provide a consistent and cross-browser-compatible way of handling events within the components of your application. Synthetic events aim to simplify event handling and make it more predictable and efficient for developers.

Here are some key characteristics and benefits of synthetic events in React:

1. **Cross-browser Compatibility:** React's synthetic events are designed to work consistently across different browsers, ensuring that you don't have to deal with the nuances and inconsistencies of native browser events. This means that you can use the same event handling code in your React application, regardless of the browser being used.

2. **Performance Optimization:** React's event system is optimized for performance. It uses event delegation and minimizes the number of event listeners attached to the DOM. Instead of attaching an event handler to every individual element, React attaches a single event listener to a higher-level container element (usually the root of the component), and events are then delegated to the appropriate component.

3. **Event Pooling:** React reuses event objects, which can lead to better performance. When an event handler is called, the event object is not immediately garbage collected. This is done to improve performance by avoiding unnecessary object creation and cleanup. However, this means that you should not access the event object asynchronously, as it will no longer be available.

4. **Consistency:** The synthetic events have the same interface as native browser events. You can access properties like `event.target`, `event.type`, and use methods like `event.preventDefault()` and `event.stopPropagation()` just as you would with native events. This makes it easier for developers who are already familiar with native event handling.

Here's an example of how synthetic events are used in React:

```jsx
class MyComponent extends React.Component {
  handleClick = (event) => {
    event.preventDefault();
    console.log('Button clicked');
  }

  render() {
    return (
      <button onClick={this.handleClick}>Click me</button>
    );
  }
}
```

In this example, the `onClick` event is a synthetic event. When the button is clicked, the `handleClick` method is called with a synthetic event object, and you can use it just like you would use a native event.

By providing this abstraction, React simplifies event handling in a way that is consistent, efficient, and compatible with various browsers, making it easier for developers to create interactive and responsive user interfaces.

11. What are forms in React? How do you create forms in React?

Form is a document that stores information of a user on a web server using interactive controls. A form contains different kinds of information such as username, password, contact number, email id, etc. Forms are a very important component of a website. Because it enables the collection of data from the user directly which makes the website much more interactive. Creating a form in React is almost similar to that of HTML if we keep it simple and just make a static form. But there is a lot of differences that come while handling the data and retrieving the form submission.

Controlled Components: In simple HTML elements like input tag, the value of the input field is changed whenever the user type. But, In React, whatever the value user types we save it in state and pass the same value to the input tag as its value, so here its value is not changed by DOM, it is controlled by react state.

We will be rendering the value inside the input box to another DOM element inside the same component using react state. Inside the App component of App.js

- Make an inputValue state with the value of an empty string.
- Set the value attribute of the input box equal to the inputValue state..
- Update the inputValue state using setState() method whenever a change occurs inside the input box.
- Set the text after 'Entered Value :' equal to inputValue state.

```
import React from 'react';
class App extends React.Component {

        state = { inputValue: '' };
        render() {
                return (
                <div>
                        <form>
                                <label> Input </label>
                                <input type="text"
                                        value={this.state.inputValue}
                                        onChange={(e) => this.setState(
                                        { inputValue: e.target.value })}/>
                        </form>
                        <div>
                                Entered Value: {this.state.inputValue}
                        </div>
                </div>
                );
        }
}

export default App;
```

12. **What is a state in React? How do we update and access state in React?(GPT)**
    In React, "state" refers to a JavaScript object that represents the dynamic data within a component. State allows you to store and manage information that can change over time, making your components dynamic and responsive to user interactions and other events. React components can have state, and when the

state changes, React automatically re-renders the component to reflect those changes in the user interface.

Here's how you define, update, and access state in React:

1. **Defining State:**
   To define state in a class-based React component, you typically initialize it in the constructor using `this.state`. State is initialized as a JavaScript object containing the initial values of the data you want to manage.

   ```jsx
   class MyComponent extends React.Component {
     constructor(props) {
       super(props);
       this.state = {
         counter: 0,
         name: 'John',
       };
     }

     // ...
   }
   ```

2. **Accessing State:**
   You can access state values within your component's `render` method or any other methods by using `this.state`. For example:

   ```jsx
   render() {
     return (
       <div>
         <p>Counter: {this.state.counter}</p>
         <p>Name: {this.state.name}</p>
       </div>
     );
   }
   ```

3. **Updating State:**
   State should not be modified directly; instead, you use the `this.setState()` method to update it. The `setState` method is used to indicate which parts of the state you want to change and what their new values should be. React will then update the component and re-render it accordingly.

```jsx
incrementCounter = () => {
  this.setState({ counter: this.state.counter + 1 });
}
```

It's important to note that `setState` is asynchronous and may not update the state immediately. To ensure you are working with the latest state, you can provide a callback function as the second argument to `setState`.

```jsx
incrementCounter = () => {
  this.setState({ counter: this.state.counter + 1 }, () => {
    console.log('Counter updated:', this.state.counter);
  });
}
```

4. **Functional Components and Hooks:**
In functional components, you can manage state using hooks, specifically the `useState` hook. Here's an example of managing state in a functional component:

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

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

  const incrementCount = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <p>Counter: {count}</p>
      <button onClick={incrementCount}>Increment</button>
    </div>
  );
}
```

In React, state management is a crucial concept for creating dynamic and interactive user interfaces. By using state, you can build components that

respond to user interactions and data changes, making your applications more engaging and user-friendly.

13. **What are props in React? How do you pass props between components?**

In React, the components need to communicate or send data to each other. Props or Properties is a special keyword in React used to pass data from one component to another. They are analogous to function arguments in JavaScript.

The important point to consider is that data with props can only be passed in a unidirectional flow, i.e., from parent component to child component.

Also, props data is read-only, which means that data coming from props should not be changed by child components.

Implementation:

ChildComponent.js

```
import React from 'react'

function ChildComponent(props) {
  return (
    <div>
      <h2>Hello,{props.text}! How are you doing?</h2>
    </div>
  )
}

export default ChildComponent;
```

ParentComponent.js

```
import React from 'react'
import ChildComponent from './ChildComponent';

function ParentComponent() {
  return (
    <div>
      <h1>I am the parent component</h1>
      <ChildComponent text='Ninja'/>
      <ChildComponent text='Geek'/>
      <ChildComponent text='ABC'/>
    </div>
  )
}

export default ParentComponent;
```

App.js

```
import ParentComponent from './ParentComponent';
function App() {
  return (
    <div className="App">
      <ParentComponent/>
    </div>
  );
}

export default App;
```

14. **What are the differences between state and props?**

Aspect	State	Props
Ownership	Owned and managed by the component itself.	Passed from parent component.
Mutability	Mutable (can be updated using `setState`).	Immutable (read-only).
Scope	Local to the component where it's defined.	Received from a parent component.
Initialization	Initialized in the constructor or `useState`.	Provided as attributes when rendering a component.
Purpose	Used to store and manage dynamic data within a component.	Used to pass data from a parent component to a child component.
Data Flow	Represents the component's internal data. Changes to state trigger component re-renders.	Represents external data passed down from parent components.
Updating	Modified using `this.setState()` in class components or the updater function in `useState` for functional components.	Cannot be modified directly within the child component; must be changed in the parent component.
Default Values	Can have initial default values when declared in a class component's constructor or using `useState`.	No default values are automatically assigned; defaults are set by the parent component using the `defaultProps` pattern.
Access	Accessed using `this.state` in class components or as an argument in functional components.	Accessed using `this.props` in class components or as an argument in functional components.
Example	```jsx	

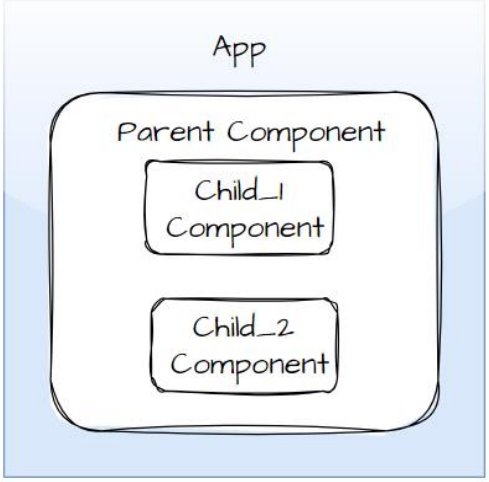
In App.js

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

```
function WelcomeMessage(props) {
return (
<div>
        <h1>Props Example</h1>
        <p>Hello, {props.name}!</p>
</div>
);
}
function Counter() {
// Declare a state variable named 'count' with an initial value of 0
const [count, setCount] = useState(0);
return (
<div>
        <h1>React Hooks Example</h1>
        <p>Count: {count}</p>
        <button onClick={() => setCount(count + 1)}>Increment</button>
        <button onClick={() => setCount(count - 1)}>Decrement</button>
</div>
);
}

function App() {
return (
<div>
        <WelcomeMessage name="Shashwat" />
        <Counter />
</div>
);
}
export default App;
```

15. **How can you embed two or more components into one in React?**
    In React, the component-based architecture allows us to create reusable and modular UI elements. Embedding two or more React components into one is the process of combining multiple smaller components into a single parent component. This allows you to create a higher-level component that encapsulates the functionality and rendering of its child components. By embedding multiple components into one, you can create more modular and reusable code structures.



Visualization of embedded components

Child Components:

```
import React from 'react';
const ChildComponent1 = () => {
  return <p>Child Component 1</p>;
};

const ChildComponent2 = () => {
  return <p>Child Component 2</p>;
};

const ChildComponent3 = () => {
  return <p>Child Component 3</p>;
};

export { ChildComponent1, ChildComponent2, ChildComponent3 };
```

Parent Component:

```
import React from 'react';
const ParentComponent = () => {
  return (
    <div>
      <ChildComponent1 />
      <ChildComponent2 />
      {/* Add more child components here */}
    </div>
  );
};

export default ParentComponent;
```

App.js:

```
import React from 'react';
import ParentComponent from './ParentComponent';

const App = () => {
  return (
    <div>
      <h1>Embedding Components</h1>
      <ParentComponent />
    </div>
  );
};

export default App;
```

16. **What are the differences between class and functional components?**

Functional Components	Class Components
A functional component is just a plain JavaScript pure function that accepts props as an argument and returns a React element(JSX).	A class component requires you to extend from React. Component and create a render function that returns a React element.
There is no render method used in functional components.	It must have the render() method returning JSX (which is syntactically similar to HTML)
Functional components run from top to bottom and once the function is returned it can't be kept alive.	The class component is instantiated and different life cycle method is kept alive and is run and invoked depending on the phase of the class component.
Also known as Stateless components as they simply accept data and display them in some form, they are mainly responsible for rendering UI.	Also known as Stateful components because they implement logic and state.
React lifecycle methods (for example, componentDidMount) cannot be used in functional components.	React lifecycle methods can be used inside class components (for example, componentDidMount).

Hooks can be easily used in functional components to make them Stateful.
Example:
`const [name,SetName]= React.useState(' ')`
	It requires different syntax inside a class component to implement hooks.
	Example:
	`constructor(props) {`
	`super(props);`
	`this.state = {name: ' '}`
	`}`

Constructors are not used.	Constructor is used as it needs to store state.

Syntax of Functional Component:

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

const FunctionalComponent = () => {
```

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

    const increase = () => {
        setCount(count + 1);
    }

    return (
        <div style={{ margin: '50px' }}>
            <h1>Welcome to Geeks for Geeks </h1>
            <h3>Counter App using Functional Component : </h3>
            <h2>{count}</h2>
            <button onClick={increase}>Add</button>
        </div>
    )
}

export default FunctionalComponent;
```

Syntax of Class Component:

```
import React, { Component } from "react";

class ClassComponent extends React.Component {
    constructor() {
        super();
        this.state = {
            count: 0
        };
        this.increase = this.increase.bind(this);
    }

    increase() {
        this.setState({ count: this.state.count + 1 });
    }

    render() {
        return (
            <div style={{ margin: '50px' }}>
                <h1>Welcome to Geeks for Geeks </h1>
                <h3>Counter App using Class Component : </h3>
                <h2> {this.state.count}</h2>
                <button onClick={this.increase}> Add</button>

            </div>
        )
    }
```

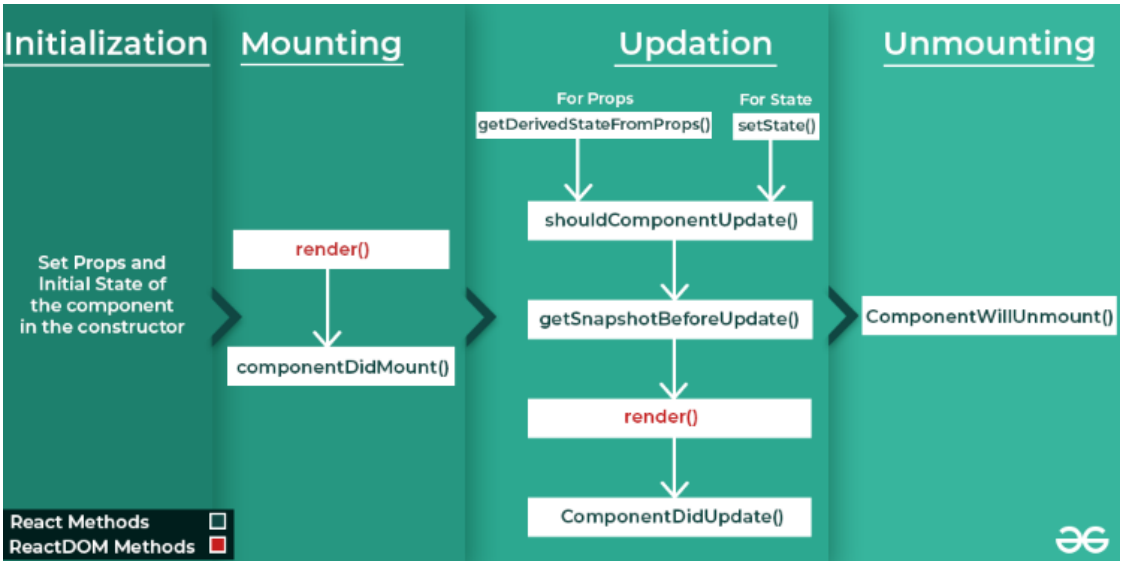
```
}

export default ClassComponent;
```

17. **Explain the Component life Cycle and its methods.**

Every React Component has a lifecycle of its own, lifecycle of a component can be defined as the series of methods that are invoked in different stages of the component's existence. The definition is pretty straightforward but what do we mean by different stages? A React Component can go through four stages of its life as follows.

Initialization: This is the stage where the component is constructed with the given Props and default state. This is done in the constructor of a Component Class.
Mounting: Mounting is the stage of rendering the JSX returned by the render method itself.
Updating: Updating is the stage when the state of a component is updated and the application is repainted.
Unmounting: As the name suggests Unmounting is the final step of the component lifecycle where the component is removed from the page.
React provides the developers with a set of predefined functions that if present are invoked around specific events in the lifetime of the component. Developers are supposed to override the functions with desired logic to execute accordingly. We have illustrated the gist in the following diagram.



Initialization

In this phase, the developer has to define the props and initial state of the component this is generally done in the constructor of the component. The following code snippet describes the initialization process.

```
class Clock extends React.Component {
   constructor(props)
```

```
    {
        // Calling the constructor of
        // Parent Class React.Component
        super(props);

        // Setting the initial state
        this.state = { date : new Date() };
    }
}
```

Mounting

Mounting is the phase of the component lifecycle when the initialization of the component is completed and the component is mounted on the DOM and rendered for the first time on the webpage. Now React follows a default procedure in the Naming Conventions of these predefined functions where the functions containing “Will” represents before some specific phase and “Did” represents after the completion of that phase. The mounting phase consists of two such predefined functions as described below.

First initialize the data and the states in the constructor

componentDidMount() Function: This function is invoked right after the component is mounted on the DOM i.e. this function gets invoked once after the render() function is executed for the first time

Updation

React is a JS library that helps create Active web pages easily. Now active web pages are specific pages that behave according to their user. For example, let’s take the GeeksforGeeks {IDE} webpage, the webpage acts differently with each user. User A might write some code in C in the Light Theme while another User may write Python code in the Dark Theme all at the same time. This dynamic behavior that partially depends upon the user itself makes the webpage an Active webpage. Now how can this be related to Updation? Updation is the phase where the states and props of a component are updated followed by some user events such as clicking, pressing a key on the keyboard, etc. The following are the descriptions of functions that are invoked at different points of the Updation phase.

getDerivedStateFromProps: getDerivedStateFromProps(props, state) is a static method that is called just before render() method in both mounting and updating phase in React. It takes updated props and the current state as arguments.

```
static getDerivedStateFromProps(props, state) {
    if(props.name !== state.name){
        //Change in props
        return{
```

```
            name: props.name
          };
        }
        return null; // No change to state
    }
```

setState() Function: This is not particularly a Lifecycle function and can be invoked explicitly at any instant. This function is used to update the state of a component. You may refer to this article for detailed information.

shouldComponentUpdate() Function: By default, every state or props update re-renders the page but this may not always be the desired outcome, sometimes it is desired that updating the page will not be repainted. The **shouldComponentUpdate()** Function fulfills the requirement by letting React know whether the component's output will be affected by the update or not. shouldComponentUpdate() is invoked before rendering an already mounted component when new props or states are being received. If returned false then the subsequent steps of rendering will not be carried out. This function can't be used in the case of forceUpdate(). The Function takes the new Props and new State as the arguments and returns whether to re-render or not.

getSnapshotBeforeUpdate() Method: The getSnapshotBeforeUpdate() method is invoked just before the DOM is being rendered. It is used to store the previous values of the state after the DOM is updated.

componentDidUpdate() Function: Similarly this function is invoked after the component is rerendered i.e. this function gets invoked once after the render() function is executed after the updation of State or Props.

Unmounting

This is the final phase of the lifecycle of the component which is the phase of unmounting the component from the DOM. The following function is the sole member of this phase.

componentWillUnmount() Function: This function is invoked before the component is finally unmounted from the DOM i.e. this function gets invoked once before the component is removed from the page and this denotes the end of the lifecycle.

18. **What are the differences between controlled and uncontrolled components in React?**

Controlled Component	Uncontrolled Component
The component is under control of the component's state.	Components are under the control of DOM.
These components are predictable as are controlled by the state of the component.	Are Uncontrolled because during the life cycle methods the data may loss
Internal state is not maintained	Internal state is maintained
It accepts the current value as props	We access the values using refs
Does not maintain its internal state.	Maintains its internal state.
Controlled by the parent component.	Controlled by the DOM itself.
Have better control on the form data and values	Has very limited control over form values and data

19. **Explain Strict Mode in React.**

    StrictMode is a React Developer Tool primarily used for **highlighting possible problems in a web application.** It **activates additional deprecation checks** and warnings for its child components. One of the reasons for its popularity is the fact that it **provides visual feedback (warning/error messages) whenever the React guidelines and recommended practices are not followed.** Just like the React Fragment, the React StrictMode Component does not render any visible UI.

    The React StrictMode can be viewed as a **helper component** that allows developers to **code efficiently and brings to their attention any suspicious code** which might have been accidentally added to the application. The StrictMode can be applied to any section of the application, not necessarily to the entire application. It is especially helpful to use while developing new codes or debugging the application.

```
return (
    <div>
      <Component1 />
      <React.StrictMode>
        <React.Fragment>
          <Component2 />
          <Component3 />
        </React.Fragment>
      </React.StrictMode>
      <Component4 />
```

```
      </div>
    );
```

In the above example, the StrictMode checks will be applicable only on Component2 and Component3 (as they are the child components of React.StrictMode). Contrary to this, Component1 and Component4 will not have any checks.

Advantages: T**he React StrictMode helps to identify and detect various warnings/errors during the development phase, namely-**

Helps to identify those components having unsafe lifecycles: Some of the legacy component lifecycle methods are considered to be unsafe to use in async applications. The React StrictMode helps to detect the use of such unsafe methods. Once enabled, it displays a list of all components using unsafe lifecycle methods as warning messages.

Warns about the usage of the legacy string ref API: Initially, there were two methods to manage refs- legacy string ref API and the callback API. Later, a third alternate method, the createRef API was added, replacing the string refs with object refs, which allowed the StrictMode to give warning messages whenever string refs are used.

Warns about the usage of the deprecated findDOMNode: Since the findDOMNode is only a one-time read API, it is not possible to handle changes when a child component attempts to render a different node (other than the one rendered by the parent component). These issues were detected by the React StrictMode and displayed as warning messages.
Since the StrictMode is a developer tool, it runs only in development mode. It does not affect the production build in any way whatsoever.
In order to identify and detect any problems within the application and show warning messages, **StrictMode renders every component inside the application twice.**

20. **What is React Router? How do we implement it?**

    React Router is a standard library for routing in React. It enables the navigation among views of various components in a React Application, allows changing the browser URL, and keeps the UI in sync with the URL. The application will contain three components: the home component, about component, and contact component.
    It is a fully-featured client and server-side routing library for React. React Router Dom is used to build single-page applications i.e. applications that have many pages or components but the page is never refreshed instead the content is dynamically fetched based on the URL. This process is called Routing and it is made possible with the help of React Router Dom.

The major advantage of react-router is that the page does not have to be refreshed when a link to another page is clicked, for example. Moreover, it is fast, very fast compared to traditional page navigation. This means that the user experience is better and the app has overall better performance.

Implementation:

1. // Installing
   npm i react-router-dom
2. // Importing
   import { BrowserRouter as Router, Routes, Route } from "react-router-dom";

3. **BrowserRouter:** BrowserRouter is a router implementation that uses the HTML5 history API(pushState, replaceState, and the popstate event) to keep your UI in sync with the URL. It is the parent component that is used to store all of the other components.
4. **Routes:** It's a new component introduced in the v6 and an upgrade of the component. The main advantages of Routes over Switch are:
   a. Relative s and s
   b. Routes are chosen based on the best match instead of being traversed in order.
5. **Route:** Route is the conditionally shown component that renders some UI when its path matches the current URL.
6. **Link:** The link component is used to create links to different routes and implement navigation around the application. It works like an HTML anchor tag.

```
return (
  <Router>
    <div className="App">
      <ul className="App-header">
      <li>
        <Link to="/">Home</Link>
      </li>
      <li>
        <Link to="/about">About Us</Link>
      </li>
      <li>
        <Link to="/contact">Contact Us</Link>
      </li>
      </ul>
    <Routes>
        <Route exact path='/' element={< Home />}></Route>
```

```
        <Route exact path='/about' element={< About />}></Route>
        <Route exact path='/contact' element={< Contact />}></Route>
      </Routes>
      </div>
    </Router>
);
```

21. **Creating a Module in Reactjs and different methods of exporting it.**
    In React, you can create a module by defining a JavaScript file that encapsulates a set of components, functions, or variables. These modules can then be exported and imported in other parts of your application to promote code organization and reusability. React supports several methods of exporting and importing modules.

    Here's an example of creating a module in React and different methods of exporting it:

    Sample Module: `mathOperations.js`

```
// Named Export: Exporting a variable
export const add = (a, b) => a + b;

// Named Export: Exporting a function
export function subtract(a, b) {
  return a - b;
}

// Named Export: Exporting a component
export const Greeting = (props) => {
  return <p>Hello, {props.name}!</p>;
}

// Default Export: Exporting a component as the default export
const DefaultComponent = (props) => {
  return <p>This is the default component.</p>;
}
export default DefaultComponent;

// Exporting multiple named exports together
export { add, subtract };
```

    In this example, we have a module named `mathOperations.js` that exports variables, functions, and a component using both named exports and a default export.

    Importing the Module: `App.js`

```
import React from 'react';
```

```
import { add, subtract, Greeting, default as DefaultComponent } from
'./mathOperations';

function App() {
  return (
    <div>
      <p>Result of addition: {add(5, 3)}</p>
      <p>Result of subtraction: {subtract(10, 4)}</p>
      <Greeting name="John" />
      <DefaultComponent />
    </div>
  );
}

export default App;
```

In `App.js`, we import various members from the `mathOperations` module using different methods:

1. Importing named exports (variables and functions) using their names within curly braces.
2. Importing a named export component (`Greeting`) using its name within curly braces.
3. Importing the default export component (`DefaultComponent`) using the `default` keyword.

Here are the methods of exporting and importing modules in React:

1. Named Exports:
  - You can export variables, functions, and components individually using `export` with their names.
  - Import these named exports by specifying the name within curly braces when importing.

2. Default Export:
  - You can export a single default component, function, or variable using `export default`.
  - Import the default export without curly braces, giving it any name you choose.

3. Combining Named Exports:
  - You can export multiple named exports together in a single statement using `export { ... }`.
  - Import them with their respective names using curly braces during import.

These methods allow you to organize your React code into modular and reusable pieces, making your codebase more maintainable and scalable.

22. React code for creating 2 component and integrating it.(write the code of necessary files

To create two React components and integrate them, you typically need at least three files:

1. The main application file (`App.js`) where you integrate the components.
2. The first component (`Component1.js`).
3. The second component (`Component2.js`).

Here's an example with the code for each file:

`Component1.js`

This is the code for the first component (`Component1.js`):

```jsx
import React from 'react';

function Component1() {
  return (
    <div>
      <h2>Component 1</h2>
      <p>This is the first component.</p>
    </div>
  );
}

export default Component1;
```

`Component2.js`

This is the code for the second component (`Component2.js`):

```jsx
import React from 'react';

function Component2() {
  return (
    <div>
      <h2>Component 2</h2>
      <p>This is the second component.</p>
    </div>
  );
}
```

```
export default Component2;
```

`App.js`
This is the code for the main application file (`App.js`) that integrates the two components:

```jsx
import React from 'react';
import Component1 from './Component1';
import Component2 from './Component2';

function App() {
  return (
    <div>
      <h1>Integration of React Components</h1>
      <Component1 />
      <Component2 />
    </div>
  );
}

export default App;
```

In this example:
- `Component1` and `Component2` are two functional components, each rendering a simple message.
- In the `App.js` file, we import `Component1` and `Component2` and use them within the `App` component.

Make sure that these files are in the same directory, and you have set up your React environment (using tools like Create React App or a custom configuration) to run the application. This example demonstrates the integration of two React components within a parent component (`App`), which is a common practice in building React applications.

23. Create functional components for performing arithmetic operations.

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

function ArithmeticOperations() {
  const [num1, setNum1] = useState(0);
  const [num2, setNum2] = useState(0);
  const [result, setResult] = useState(0);
```

```
const handleNum1Change = (event) => {
  setNum1(Number(event.target.value));
};

const handleNum2Change = (event) => {
  setNum2(Number(event.target.value));
};

const add = () => {
  setResult(num1 + num2);
};

const subtract = () => {
  setResult(num1 - num2);
};

const multiply = () => {
  setResult(num1 * num2);
};

const divide = () => {
  if (num2 === 0) {
    setResult('Error: Division by zero');
  } else {
    setResult(num1 / num2);
  }
};

return (
  <div>
    <h1>Arithmetic Operations</h1>
    <input
      type="number"
      value={num1}
      onChange={handleNum1Change}
    />
    <input
      type="number"
      value={num2}
      onChange={handleNum2Change}
    />
    <div>
      <button onClick={add}>Add</button>
      <button onClick={subtract}>Subtract</button>
```

```
        <button onClick={multiply}>Multiply</button>
        <button onClick={divide}>Divide</button>
      </div>
      <p>Result: {result}</p>
    </div>
  );
}

export default ArithmeticOperations;
```

In this code:
1. We import `React` and `useState` from 'react'.
2. We create a functional component called `ArithmeticOperations`.
3. Inside the component, we use the `useState` hook to manage state for `num1`, `num2`, and `result`.

4. We define event handlers (`handleNum1Change` and `handleNum2Change`) to update `num1` and `num2` when the user enters values in the input fields.
5. We define four arithmetic functions: `add`, `subtract`, `multiply`, and `divide`. These functions calculate the result based on the user's input and update the `result` state accordingly.
6. In the `return` statement, we render input fields for `num1` and `num2,` along with buttons to perform arithmetic operations.
7. When a button is clicked, the corresponding arithmetic operation is executed, and the result is displayed below the buttons.

This functional component provides a basic UI for performing arithmetic operations in a React application. Users can enter numbers, choose an operation, and see the result.

24. Class component for login

```
import React, { Component } from 'react';

class Login extends Component {
  constructor(props) {
    super(props);

    // Initialize the component's state to hold user input
    this.state = {
      username: '',
      password: '',
      loggedIn: false
    };
  }
```

```
// Event handler for input changes
handleInputChange = (event) => {
  const { name, value } = event.target;

  // Update the state with the new input value
  this.setState({
    [name]: value
  });
};

// Event handler for the form submission
handleFormSubmit = (event) => {
  event.preventDefault();

  // Simulate a login check (e.g., with hardcoded values)
  if (this.state.username === 'user' && this.state.password === 'password') {
    this.setState({ loggedIn: true });
  } else {
    alert('Invalid credentials. Please try again.');
  }
};

render() {
  return (
    <div>
      <h1>Login</h1>
      {this.state.loggedIn ? (
        <p>Welcome, {this.state.username}!</p>
      ) : (
        <form onSubmit={this.handleFormSubmit}>
          <div>
            <label>Username:</label>
            <input
              type="text"
              name="username"
              value={this.state.username}
              onChange={this.handleInputChange}
            />
          </div>
          <div>
            <label>Password:</label>
            <input
              type="password"
              name="password"
              value={this.state.password}
```

```
            onChange={this.handleInputChange}
          />
        </div>
        <div>
          <button type="submit">Login</button>
        </div>
      </form>
    )}
  </div>
 );
 }
}

export default Login;
```

Now, let's break down this code:

1. We import React and `Component` from 'react'. We are creating a class component called `Login`.
2. In the `constructor`, we initialize the component's state with `username`, `password`, and `loggedIn` properties. These properties will hold user input and determine whether the user is logged in.
3. We define two event handlers:
   - `handleInputChange`: This function handles changes in the input fields. It uses the `name` attribute to dynamically update the corresponding state property (`username` or `password`).
   - `handleFormSubmit`: This function handles form submission. It prevents the default form submission behavior, checks if the entered username and password match a hardcoded value, and either logs in the user or displays an alert for invalid credentials.
4. In the `render` method, we conditionally render either a welcome message or the login form based on the `loggedIn` state property.
5. If the user is logged in, we display a welcome message. Otherwise, we render a form with input fields for the username and password. When the form is submitted, the `handleFormSubmit` function is called.

This class component provides a basic example of a login form in React. When a user enters the correct credentials (username: 'user', password: 'password'), they will be logged in and greeted with a welcome message. Otherwise, an alert will inform them of the incorrect login.

25. **Code for props and state**

In App.js

```
import React, { useState } from "react";
function WelcomeMessage(props) {
return (
```

```
<div>
        <h1>Props Example</h1>
        <p>Hello, {props.name}!</p>
</div>
);
}
function Counter() {
// Declare a state variable named 'count' with an initial value of 0
const [count, setCount] = useState(0);
return (
<div>
        <h1>React Hooks Example</h1>
        <p>Count: {count}</p>
        <button onClick={() => setCount(count + 1)}>Increment</button>
        <button onClick={() => setCount(count - 1)}>Decrement</button>
</div>
);
}

function App() {
return (
<div>
        <WelcomeMessage name="Shashwat" />
        <Counter />
</div>
);
}
export default App;
```

26. Event

```
import React, { Component } from 'react';

class EventHandlingExample extends Component {
  constructor() {
    super();

    // Initialize the component's state to hold a message
    this.state = {
      message: 'Click the button!',
    };
  }

  // Event handler for the button click
  handleClick = () => {
    // Update the message when the button is clicked
```

```
    this.setState({
      message: 'Button was clicked!',
    });
  };

  render() {
    return (
      <div>
        <h1>Event Handling in React</h1>
        <p>{this.state.message}</p>
        <button onClick={this.handleClick}>Click me</button>
      </div>
    );
  }
}

export default EventHandlingExample;
```

In this code:

1. We import `React` and `Component` from 'react'.
2. We create a class component called `EventHandlingExample`.
3. In the component's constructor, we initialize the state with a message.

4. We define an event handler method `handleClick` that updates the state when the button is clicked.
5. In the `render` method, we display the message and a button element. We attach the `onClick` event listener to the button, which calls the `handleClick` method when the button is clicked.
6. When the button is clicked, the state is updated, and the new message is displayed in the component.

27. React folder structure. use of package.json,src etc.

Folder Structure for React Project:

1) Assets Folder
2) Layouts Folder
3) Components Folder
4) Pages Folder
5) Middleware Folder
6) Routes Folder
7) Config Folder
8) Services Folder
9) Utils Folder

- **Assets Folder**

As the name says, it contains assets of our project. It consists of images and styling files. Here we can store our global styles. We are centralizing the project so we can store the page-based or component-based styles over here. But we can even keep style according to the pages folder or component folder also. But that depends on developer comfortability.

- **Layouts Folder**

As the name says, it contains layouts available to the whole project like header, footer, etc. We can store the header, footer, or sidebar code here and call it.

- **Components Folder**

Components are the building blocks of any react project. This folder consists of a collection of UI components like buttons, modals, inputs, loader, etc., that can be used across various files in the project. Each component should consist of a test file to do a unit test as it will be widely used in the project.

- **Pages Folder**

The files in the pages folder indicate the route of the react application. Each file in this folder contains its route. A page can contain its subfolder. Each page has its state and is usually used to call an async operation. It usually consists of various components grouped.

- **Middleware Folder**

This folder consists of middleware that allows for side effects in the application. It is used when we are using redux with it. Here we keep all our custom middleware.

- **Routes Folder**

This folder consists of all routes of the application. It consists of private, protected, and all types of routes. Here we can even call our sub-route.

- **Config Folder**

This folder consists of a configuration file where we store environment variables in config.js. We will use this file to set up multi-environment configurations in your application.

- **Services Folder**

This folder will be added if we use redux in your project. Inside it, there are 3 folders named actions, reducers, and constant subfolders to manage states. The actions and reducers will be called in almost all the pages, so create actions, reducers & constants according to pages name.

- **Utils Folder**

Utils folder consists of some repeatedly used functions that are commonly used in the project. It should contain only common js functions & objects like dropdown options, regex condition, data formatting, etc.

Advance React:

28. **What is React Hook?different hooks supported by React? Explain useState() and useEffect()(GPT)**

React Hooks are functions that allow you to use state and other React features in functional components. They were introduced in React 16.8 to provide a more

straightforward and cleaner way of managing state and side-effects in functional components, which were previously only possible in class components. React Hooks make it easier to reuse stateful logic across components and enhance the readability and maintainability of your code.

There are several built-in hooks in React, and two of the most fundamental ones are `useState()` and `useEffect()`.

1. useState():
- `useState()` is a hook that enables you to add state to functional components. It takes an initial state value as an argument and returns an array with two elements: the current state value and a function to update it.

Example:
```jsx
import React, { useState } from 'react';

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

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```
In the example above, `useState(0)` initializes the `count` state variable to 0. The `setCount` function can be used to update the state when the "Increment" button is clicked.

2. useEffect():
- `useEffect()` is a hook for managing side effects in your component. It allows you to perform tasks like data fetching, DOM manipulation, or subscribing to external events. It takes two arguments: a function to run the effect and an optional array of dependencies.

Example:
```jsx
import React, { useState, useEffect } from 'react';

function ExampleComponent() {
  const [data, setData] = useState(null);
```

```
  useEffect(() => {
    // Fetch data when the component mounts
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(result => setData(result))
      .catch(error => console.error(error));
  }, []); // Empty dependency array means this effect runs once on mount

  return (
    <div>
      {data ? (
        <p>Data: {data}</p>
      ) : (
        <p>Loading data...</p>
      )}
    </div>
  );
}
```

In this example, the `useEffect` hook is used to fetch data when the component mounts. The empty dependency array ensures that the effect runs only once when the component is mounted.

React also provides other hooks like `useContext`, `useReducer`, `useRef`, and more, which serve various purposes in functional components. These hooks help manage different aspects of your application's state and logic while maintaining the benefits of functional components.

In summary, React Hooks, such as `useState()` and `useEffect()`, are powerful tools that make it easier to manage state and side effects in functional components. They simplify your code and improve the overall development experience in React.

29. React hooks?(similar as above)(GPT)

React Hooks are a fundamental feature introduced in React 16.8 that revolutionized the way developers work with state and side effects in React functional components. They provide a more elegant and flexible approach compared to class components, allowing developers to encapsulate and reuse logic within their components, resulting in cleaner, more maintainable code. Here's a detailed explanation of React Hooks:

1. Introduction to Hooks (1 mark):

React Hooks are a set of functions provided by React that allow functional components to manage state, side effects, and other React features that were

traditionally exclusive to class components. They were introduced to simplify the development process and promote the use of functional components.

2. State Management with `useState()` (2 marks):
   - The `useState()` hook is used for adding state to functional components. It takes an initial state value as an argument and returns an array containing two elements: the current state value and a function to update it.
   - This hook makes it easy to manage component-local state without writing class-based code. It's a fundamental building block for creating interactive UIs in React.

3. Example of `useState()` (1 mark):
   ```jsx
   import React, { useState } from 'react';

   function Counter() {
     const [count, setCount] = useState(0);
     return (
       <div>
         <p>Count: {count}</p>
         <button onClick={() => setCount(count + 1)}>Increment</button>
       </div>
     );
   }
   ```

4. Side Effect Management with `useEffect()` (2 marks):
   - The `useEffect()` hook enables the management of side effects in functional components. It's used for tasks such as data fetching, DOM manipulation, and event subscriptions.
   - It accepts two arguments: a function that defines the side effect and an optional array of dependencies. The effect runs when the component renders and re-renders if any of the dependencies change.

5. Example of `useEffect()` (1 mark):
   ```jsx
   import React, { useState, useEffect } from 'react';

   function ExampleComponent() {
     const [data, setData] = useState(null);
     useEffect(() => {
       fetch('https://api.example.com/data')
         .then(response => response.json())
         .then(result => setData(result))
         .catch(error => console.error(error));
   ```

```
  }, []); // Empty dependency array means the effect runs once on mount
}
```

6. Benefits of React Hooks (2 marks):
   - Improved Code Readability: Hooks promote the organization of component logic based on what it does (state, effect, context, etc.) rather than lifecycle methods.
   - Reusability: Hooks facilitate the creation of custom hooks, making it easy to share and reuse logic across different components.
   - Simplicity: Hooks eliminate the need for classes and reduce the cognitive load on developers by making the code more straightforward.
   - Easier Testing: Functional components with hooks are easier to test due to their improved modularity.

7. Custom Hooks (1 mark):
   Custom hooks are functions that use one or more built-in hooks to create reusable logic. They allow developers to abstract complex behavior into a simple, shareable function, further enhancing code maintainability and reuse.

8. Additional Hooks (1 mark):
   React provides various other hooks like `useContext`, `useReducer`, `useRef`, and `useMemo` to address specific use cases in your applications. These hooks collectively provide a comprehensive set of tools for building React applications.

In conclusion, React Hooks are a revolutionary addition to the React library that empowers developers to write more readable, maintainable, and reusable code in functional components. They simplify state management and side effect handling, making React development a more enjoyable and efficient experience.

30. What is MVC and Flux architecture?

The Model-View-Controller (MVC) framework is an architectural/design pattern that separates an application into three main logical components Model, View, and Controller. Each architectural component is built to handle specific development aspects of an application. It isolates the business logic and presentation layer from each other. It was traditionally used for desktop graphical user interfaces (GUIs). Nowadays, MVC is one of the most frequently used industry-standard web development frameworks to create scalable and extensible projects. It is also used for designing mobile apps.

MVC was created by Trygve Reenskaug. The main goal of this design pattern was to solve the problem of users controlling a large and complex data set by splitting a large application into specific sections that all have their own purpose.

Features of MVC :

1. It provides a clear separation of business logic, Ul logic, and input logic.
2. It offers full control over your HTML and URLs which makes it easy to design web application architecture.
3. It is a powerful URL-mapping component using which we can build applications that have comprehensible and searchable URLs.
4. It supports Test Driven Development (TDD).

Components of MVC :

The MVC framework includes the following 3 components:

1. Controller
2. Model
3. View

Controller:

The controller is the component that enables the interconnection between the views and the model so it acts as an intermediary. The controller doesn't have to worry about handling data logic, it just tells the model what to do. It processes all the business logic and incoming requests, manipulates data using the Model component, and interact with the View to render the final output.

View:

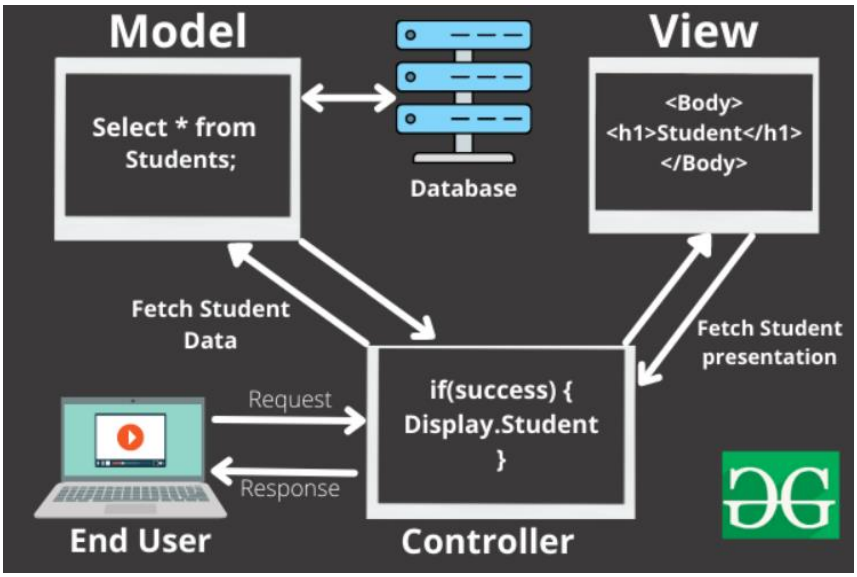
The View component is used for all the UI logic of the application. It generates a user interface for the user. Views are created by the data which is collected by the model component but these data aren't taken directly but through the controller. It only interacts with the controller.

Model:

The Model component corresponds to all the data-related logic that the user works with. This can represent either the data that is being transferred between the View and Controller components or any other business logic-related data. It can add or retrieve data from the database. It responds to the controller's request because the controller can't interact with the database by itself. The model interacts with the database and gives the required data back to the controller.

Working of the MVC framework with an example:

Let's imagine an end-user sends a request to a server to get a list of students studying in a class. The server would then send that request to that particular controller that handles students. That controller would then request the model that handles students to return a list of all students studying in a class.



The model would query the database for the list of all students and then return that list back to the controller. If the response back from the model was successful, then the controller would ask the view associated with students to return a presentation of the list of students. This view would take the list of students from the controller and render the list into HTML that can be used by the browser.

The controller would then take that presentation and returns it back to the user. Thus ending the request. If earlier the model returned an error, the controller would handle that error by asking the view that handles errors to render a presentation for that particular error. That error presentation would then be returned to the user instead of the student list presentation.

As we can see from the above example, the model handles all of the data. The view handles all of the presentations and the controller just tells the model and view of what to do. This is the basic architecture and working of the MVC framework.

The MVC architectural pattern allows us to adhere to the following design principles:

1. Divide and conquer. The three components can be somewhat independently designed.

2. Increase cohesion. The components have stronger layer cohesion than if the view and controller were together in a single UI layer.

3. Reduce coupling. The communication channels between the three components are minimal and easy to find.

4. Increase reuse. The view and controller normally make extensive use of reusable components for various kinds of UI controls. The UI, however, will become application-specific, therefore it will not be easily reusable.

5. Design for flexibility. It is usually quite easy to change the UI by changing the view, the controller, or both.

Advantages of MVC:

1. Codes are easy to maintain and they can be extended easily.
2. The MVC model component can be tested separately.
3. The components of MVC can be developed simultaneously.
4. It reduces complexity by dividing an application into three units. Model, view, and controller.
5. It supports Test Driven Development (TDD).
6. It works well for Web apps that are supported by large teams of web designers and developers.
7. This architecture helps to test components independently as all classes and objects are independent of each other
8. Search Engine Optimization (SEO) Friendly.

Disadvantages of MVC:

1. It is difficult to read, change, test, and reuse this model
2. It is not suitable for building small applications.
3. The inefficiency of data access in view.
4. The framework navigation can be complex as it introduces new layers of abstraction which requires users to adapt to the decomposition criteria of MVC.
5. Increased complexity and Inefficiency of data

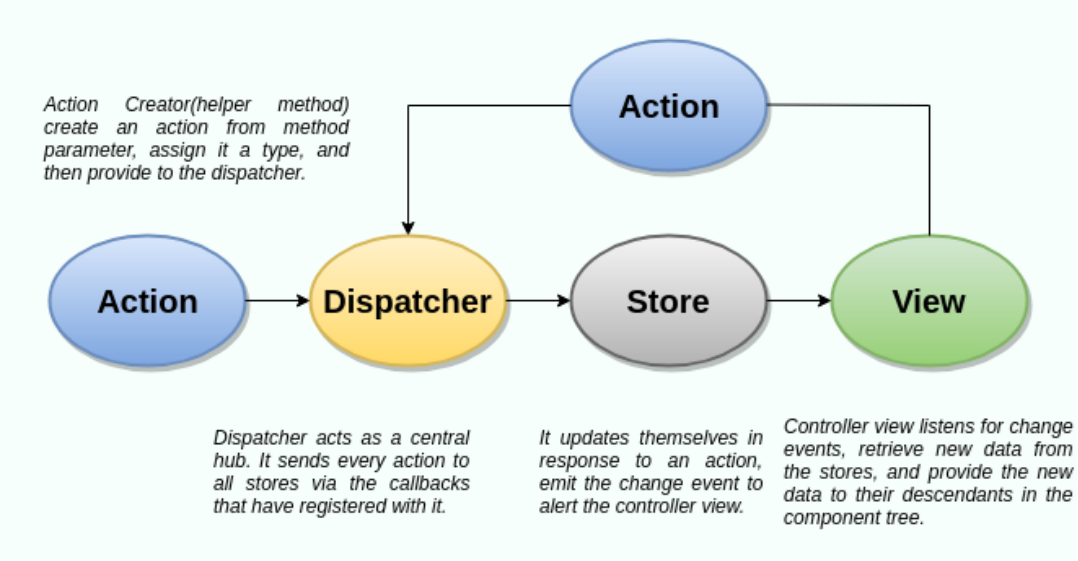
Flux is an application architecture that Facebook uses internally for building the client-side web application with React. It is not a library nor a framework. It is a kind of architecture that complements React as view and follows the concept of Unidirectional Data Flow model. It is useful when the project has dynamic data, and we need to keep the data updated in an effective manner. It reduces the runtime errors.

Flux applications have three major roles in dealing with data:

1. Dispatcher
2. Stores
3. Views (React components)

Here, you should not be confused with the Model-View-Controller (MVC) model. Although, Controllers exists in both, but Flux controller-views (views) found at the top of the hierarchy. It retrieves data from the stores and then passes this data down to their children. Additionally, action creators - dispatcher helper methods used to describe all changes that are possible in the application. It can be useful as a fourth part of the Flux update cycle.

Structure and Data Flow



In Flux application, data flows in a single direction(unidirectional). This data flow is central to the flux pattern. The dispatcher, stores, and views are independent nodes with inputs and outputs. The actions are simple objects that contain new data and type property. Now, let us look at the various components of flux architecture one by one.

Dispatcher

It is a central hub for the React Flux application and manages all data flow of your Flux application. It is a registry of callbacks into the stores. It has no real intelligence of its own, and simply acts as a mechanism for distributing the actions to the stores. All stores register itself and provide a callback. It is a place which handled all events that modify the store. When an action creator provides a new action to the dispatcher, all stores receive that action via the callbacks in the registry.

The dispatcher's API has five methods. These are:

SN	Methods	Descriptions
1.	register()	It is used to register a store's action handler callback.
2.	unregister()	It is used to unregisters a store's callback.
3.	waitFor()	It is used to wait for the specified callback to run first.
4.	dispatch()	It is used to dispatches an action.
5.	isDispatching()	It is used to checks if the dispatcher is currently dispatching an action.

Stores

It primarily contains the application state and logic. It is similar to the model in a traditional MVC. It is used for maintaining a particular state within the application, updates itself in response to an action, and emits the change event to alert the controller view.

Views

It is also called controller-views. It is located at the top of the chain to store the logic to generate actions and receive new data from the store. It is a React component listen to change events and receives the data from the stores and re-render the application.

Actions
The dispatcher method allows us to trigger a dispatch to the store and include a payload of data, which we call an action. It is an action creator or helper method that passes the data to the dispatcher.

Advantage of Flux

1. It is a unidirectional data flow model which is easy to understand.
2. It is open source and more of a design pattern than a formal framework like MVC architecture.
3. The flux application is easier to maintain.
4. The flux application parts are decoupled.

31. Differentiate MVC and Flux

SN	MVC	FLUX
1.	It was introduced in 1976.	It was introduced just a few years ago.
2.	It supports Bi-directional data Flow model.	It supports Uni-directional data flow model.
3.	In this, data binding is the key.	In this, events or actions are the keys.
4.	It is synchronous.	It is asynchronous.
5.	Here, controllers handle everything(logic).	Here, stores handle all logic.
6.	It is hard to debug.	It is easy to debug because it has common initiating point: Dispatcher.
7.	It is difficult to understand as the project size increases.	It is easy to understand.
8.	Its maintainability is difficult as the project scope goes huge.	Its maintainability is easy and reduces runtime errors.
9.	Testing of application is difficult.	Testing of application is easy.
10.	Scalability is complex.	It can be easily scalable.

32. What are the uses of Refs? (GPT)

Refs in React: A Detailed Explanation

Refs in React are a feature that allow you to access and interact with the DOM (Document Object Model) and to directly manipulate React elements. They provide a way to reference and work with individual elements or components within a React application, which can be necessary for various use cases. Below is a comprehensive explanation of what refs are and how they work in React:

1. Introduction to Refs (1 mark):
Refs are a feature in React that provide a direct way to access and interact with DOM elements and React components. They allow you to bypass React's virtual

DOM and directly modify the DOM or get a reference to a React component instance.

2. Creating Refs (1 mark):
To create a ref in React, you typically use the `React.createRef()` function or the `useRef` hook (if you're working with functional components). For example:

Using `React.createRef()` in a class component:
```jsx
class MyComponent extends React.Component {
  myRef = React.createRef();
}
```

Using `useRef` in a functional component:
```jsx
import React, { useRef } from 'react';

function MyComponent() {
  const myRef = useRef();
}
```

3. Attaching Refs to DOM Elements (2 marks):
Once you've created a ref, you can attach it to a DOM element by using the `ref` attribute. This allows you to directly access and manipulate the DOM element in your React component:

```jsx
class MyComponent extends React.Component {
  myRef = React.createRef();

  componentDidMount() {
    this.myRef.current.focus();
  }

  render() {
    return <input ref={this.myRef} />;
  }
}
```

In this example, we attached the `myRef` to an input element and focused on it when the component mounted.

4. Accessing React Component Instances (2 marks):
Refs are not limited to just DOM elements; you can also use them to reference and interact with React component instances. This can be helpful when you need to call methods or access properties of a child component:

```jsx
class ParentComponent extends React.Component {
  childRef = React.createRef();

  handleClick = () => {
    this.childRef.current.doSomething();
  }

  render() {
    return (
      <div>
        <ChildComponent ref={this.childRef} />
        <button onClick={this.handleClick}>Call Child Method</button>
      </div>
    );
  }
}
```

In this example, we have a `ParentComponent` that can call a method on a `ChildComponent` using a ref.

5. When to Use Refs (2 marks):
Refs should be used sparingly in React, as they break the typical unidirectional data flow and encapsulation of React components. You should consider using refs when:
- You need to focus or select a DOM element.
- You need to trigger imperative animations.
- You need to integrate with non-React libraries.
- You need to access and manipulate a React component instance directly.

6. Caveats and Considerations (2 marks):
It's important to understand that using refs can lead to potential pitfalls and issues if not used carefully. Some considerations include:

- Refs can be null if used before the component is mounted.
- Refs can introduce tight coupling between components.
- Overusing refs can make the code less predictable and harder to maintain.

7. Forwarding Refs (1 mark):

If you want to pass a ref from a parent component to a child component, you can use the `React.forwardRef` function. This allows you to maintain a clean separation between the parent and child components while still accessing the child's DOM element or instance.

8. Conclusion (1 mark):
Refs in React are a powerful feature that should be used with caution. They provide a way to interact with the DOM and React component instances directly, enabling you to address specific use cases where the typical React data flow doesn't suffice. However, they should be used judiciously to maintain the advantages of React's declarative and component-based approach.

33. **Program on Hook example (GPT)**

Certainly! Here's a simple React program that demonstrates the use of hooks, specifically the `useState()` hook, in a functional component. This program creates a counter that you can increment and decrement:

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

function Counter() {
  // Declare a state variable 'count' with an initial value of 0
  const [count, setCount] = useState(0);

  // Event handlers to increment and decrement the count
  const increment = () => {
    setCount(count + 1);
  };

  const decrement = () => {
    setCount(count - 1);
  };

  return (
    <div>
      <h2>Counter Example using React Hooks</h2>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
}

export default Counter;
```

In this program:
1. We import `React` and the `useState` hook from the 'react' library.
2. Inside the `Counter` functional component, we use `useState(0)` to initialize a state variable called `count` with an initial value of 0.
3. We define two event handler functions, `increment` and `decrement`, which call `setCount` to update the `count` state when the "Increment" and "Decrement" buttons are clicked.
4. In the component's `return` statement, we display the current value of `count`, along with "Increment" and "Decrement" buttons. When you click these buttons, the state is updated, and the component re-renders with the new count value.
You can use this `Counter` component in your React application to understand how the `useState()` hook allows you to manage and update state in a functional component.

34. Program on Event Handling,REf,etc. (GPT)

Here's a React program that demonstrates event handling and the use of `refs` to access and manipulate DOM elements. In this example, we'll create a simple "Add to List" feature where you can enter items and add them to a list.

```jsx
import React, { useState, useRef } from 'react';

function App() {
  const [items, setItems] = useState([]);
  const itemInputRef = useRef(null);

  const addItem = () => {
    const newItem = itemInputRef.current.value;
    if (newItem) {
      setItems([...items, newItem]);
      itemInputRef.current.value = ''; // Clear the input field
    }
  };

  return (
    <div>
      <h1>Event Handling and Refs Example</h1>
      <div>
        <input type="text" ref={itemInputRef} placeholder="Enter an item" />
        <button onClick={addItem}>Add to List</button>
      </div>
      <ul>
        {items.map((item, index) => (
```

```
        <li key={index}>{item}</li>
      ))}
    </ul>
  </div>
 );
}

export default App;
```

In this program:
1. We import `React`, `useState`, and `useRef` from the 'react' library.
2. Inside the `App` component, we declare a state variable `items` to store the list of items entered by the user and a ref `itemInputRef` to access the input field element.
3. The `addItem` function is defined to add a new item to the list. It gets the value from the `itemInputRef`, adds it to the `items` state, and then clears the input field.
4. In the component's `return` statement, we render an input field, a button, and a list. When the button is clicked, the `addItem` function is called. The list is generated based on the items in the `items` state.

This program demonstrates how to use event handling (button click) and `refs` (to access and manipulate the input field) in a React application. When you enter an item and click "Add to List," it's added to the list displayed below.

Nodejs and Express:

1. **Node Js features,advantages and disadvantages**
   Features of NodeJS:
1. **Asynchronous and Event-Driven:** The Node.js library's APIs are all asynchronous (non-blocking) in nature. A server built with Node.JS never waits for data from an API. After accessing an API, the server moves on to the next one. In order to receive and track responses of previous API requests, it uses a notification mechanism called Events.
2. **Single-Threaded:** Node.js employs a single-threaded architecture with event looping, making it very scalable. In contrast to typical servers, which create limited threads to process requests, the event mechanism allows the node.js server to reply in a non-blocking manner and makes it more scalable. When compared to traditional servers like Apache HTTP Server, Node.js uses a single-threaded program that can handle a considerably larger number of requests.
3. **Scalable:** NodeJs addresses one of the most pressing concerns in software development: scalability. Nowadays, most organizations demand scalable software. NodeJs can also handle concurrent requests efficiently. It has a cluster module that manages load balancing for all CPU cores that are active. The capability of NodeJs to partition applications horizontally is its most appealing feature. It achieves this through the use of child processes. This allows the organizations to provide distinct app versions to different target audiences, allowing them to cater to client preferences for customization.

4. **Quick execution of code:** Node.js makes use of the V8 JavaScript Runtime motor, which is also used by Google Chrome. Hub provides a wrapper for the JavaScript motor, which makes the runtime motor faster. As a result, the preparation of requests inside Node.js becomes faster as well.
5. **Cross-platform compatibility:** NodeJS may be used on a variety of systems, including Windows, Unix, Linux, Mac OS X, and mobile devices. It can be paired with the appropriate package to generate a self-sufficient executable.
6. **Uses JavaScript:** JavaScript is used by the Node.js library, which is another important aspect of Node.js from the engineer's perspective. Most of the engineers are already familiar with JavaScript. As a result, a designer who is familiar with JavaScript will find that working with Node.js is much easier.
7. **Fast data streaming:** When data is transmitted in multiple streams, processing them takes a long time. Node.js processes data at a very fast rate. It processes and uploads a file simultaneously, thereby saving a lot of time. As a result, NodeJs improves the overall speed of data and video streaming.
8. **No Buffering:** In a Node.js application, data is never buffered.

Advantages of Using Node.js:

1. Speed: One of the key benefits of Node.js development is its speed, which makes it a great choice for dynamic applications. The runtime environment of Node.js is based on an event loop that handles multiple concurrent requests easily and quickly, allowing you to scale your application with ease.

Node.js also uses non-blocking I/O (asynchronous IO), which means that instead of waiting for each request to finish before processing another one, it returns immediately after receiving the response from a previous request—allowing you to handle many more requests at once without any noticeable delay in performance or responsiveness!

2. Productivity: Node.js is a highly productive platform for developing web applications. It's a single-threaded, event-driven environment, which makes it ideal for real-time applications such as chat and video streaming. This makes Node.js an ideal choice for building highly interactive websites with Ajax functionality as well as Divi Builder extensions that allow you to build custom themes without having to write any PHP code or CSS file types (like .css).

3. Error handling: Node.js has a built-in error-handling mechanism that allows you to catch errors at runtime and do something with them. This is similar to the try/catch mechanism of Java and C++, but in Node, it's easier than ever because there are no exceptions! Errors are just thrown as an event that can be caught by your code. You can also use "error" objects instead of throwing errors directly into other functions or methods if you need more control over what happens when an error occurs (e.g., logging).

4. Cost-effectiveness: There are several benefits to using Node.js development services, including cost-effectiveness. The main reason why companies choose this technology is because of its cost savings and time savings.
Cost Savings: With a comprehensive cloud solution from NodeStack, you can save money in many ways including:

• Lowering your IT costs by reducing the amount of hardware required for server deployments;
• Decreasing operating expenses related to software licensing;
• Eliminating maintenance costs associated with upgrading or patching existing applications or servers

5. Faster development: Node.js is a platform for building fast websites and web apps. It's also the most popular platform for building microservices, which means you can use it to build applications that have many small parts that work together as one system.

Node.js allows developers to create high-performance applications in a fraction of the time required by other languages and platforms like Java or C++ (which are both very good at handling large amounts of data). This makes it easy to launch new products quickly while still maintaining quality control over each part of your application since each component will be tested individually before being added to the full stack

6. Better performance in slow network environments: As you might expect, the single-threaded nature of node.js makes it more suitable for handling slow network connections and other tasks that require a lot of processing power. The non-blocking I/O model used by NodeJS allows it to handle many concurrent connections without waiting for other threads or processes to finish their work before continuing on with yours. This makes your application run faster than if you were using a language like PHP or Java which are multi-threaded and can only do one thing at a time (or not even that).

7. Highly scalable applications: Node.js is an open-source, cloud-based platform that offers many benefits to developers. It is highly scalable and responds quickly to changes in the data layer, allowing you to build large applications with ease. The following are some of the ways you can scale your Node.js application:

• Use a cloud provider such as Amazon Web Services (AWS) or Microsoft Azure Cloud Platform (MCP). These providers offer dedicated resources that will help reduce costs while increasing performance—and they're free!

• Run your code on-premises using VMware vCloud Air or EMC VNXe environment management software solutions (EMSS). This method allows you to

access VM instances within VMWare environments without having any additional hardware requirements because all required hardware has been preconfigured into them already by their respective manufacturers

However, keep in mind that this method may not be suitable for high-performance applications since it requires significant resources from both servers which could lead up to scalability issues if not properly managed correctly."

8. Full-stack JS developer requirement: Node.js is a full-stack framework, which means it can be used in multiple programming languages and frameworks. It's not a replacement for Java or PHP, but rather an alternative to them if you want to build web applications that need to scale on the server side (such as e-commerce websites). Node.js isn't meant to replace existing languages like Python or Ruby; instead, it offers an easier way for developers who have never used JavaScript before but still want access to some of its features like asynchronous processing and non-blocking I/O operations—which means no more waiting around while your program runs!

9. Node.js is a reliable, high-performing, and highly flexible platform for developing dynamic web apps:
Node.js is a fast and scalable platform for developing web applications, used to build real-time apps, server-side, and networking applications. It is an open-source, cross-platform runtime environment that allows you to build scalable networked applications using JavaScript.

The disadvantages of Using Node.js:
1. Asynchronous Programming Model: Asynchronous programming is a form of programming in which program execution proceeds without waiting for all program elements to be processed. In asynchronous programming, program elements that can run independently of other elements are processed in parallel, and program elements that cannot run independently are processed sequentially.

2. Unstable API: An unstable API is one that is subject to change at any time. This can be a problem for developers who are relying on the API to remain constant in order to maintain their own software. An unstable API can also lead to software that is difficult to maintain and upgrade.

If you're using node.js, you need to be aware that the API is unstable. That means that it's subject to change at any time, and you could potentially break your code if you're not careful. It's important to keep up to date with the latest changes and to be prepared for breaking changes.

3. Dealing with Relational Database: A relational database is a database that stores data in tables that are related to each other. This is in contrast to a non-relational database, which does not have this concept of relations between tables. Relational databases are the most widely used type of database, and they are well-suited for many applications. However, they can be more difficult to work with than non-relational databases, and this can be a pain point for developers.

Node.js is a JavaScript runtime that is well-suited for working with relational databases. Node.js also has a module called the mysql module that allows you to connect to a MySQL database and perform database operations.

However, this can also be a pain point, as some libraries are more difficult to use than others. They can be more difficult to work with than non-relational databases. Node.js can make working with relational databases easier, but it can also be a pain point.

Still, If you want non-relational persistence just simply de-normalize a relational database. If you use a relational DB with Node.js I recommend this ORM.

4. Lacks a Strong Library Support System: Node.js lacks strong library support. Because Node.js is relatively new, there are fewer high-quality libraries available for it compared to other programming languages. This can make it more difficult to find the right library for a given task, and it can also make it more difficult to get started with Node.js development. Additionally, the libraries that are available for Node.js often have fewer features than their counterparts in other languages. This makes it difficult for the developers to even implement the common programming tasks using Node.js also it makes development more challenging, and it can also lead to more bugs and issues in production code.

5. Not Suited for CPU-intensive Tasks: Node.js applications are single-threaded, which means that they can only use one CPU core at a time. This can be a bottleneck for applications that are CPU-intensive.

As we know, Node.js is a runtime environment that executes JavaScript on the server side. Being a front-end programming language, JavaScript uses a single thread to process tasks fast. Threading is not required for it to work, because tasks in JavaScript are lightweight and take little CPU.

A non-blocking input/output model means that Node.js answers the client call to start a request and waits for I/O tasks to complete in the background while executing the rest and then returns to I/O tasks using callbacks. Processing requests asynchronously, Node executes JS code on its single thread on an event basis. That is what is called an event loop. This event loop problem occurs when Node.js receives a CPU-intensive task.

Whenever a heavy request comes to the event loop, Node.js will set all the CPU available to process it in the first queue, and then execute other pending requests in a queue. As a result, it will bottleneck in processing. That's why it's not suitable for High intensive CPU tasks.

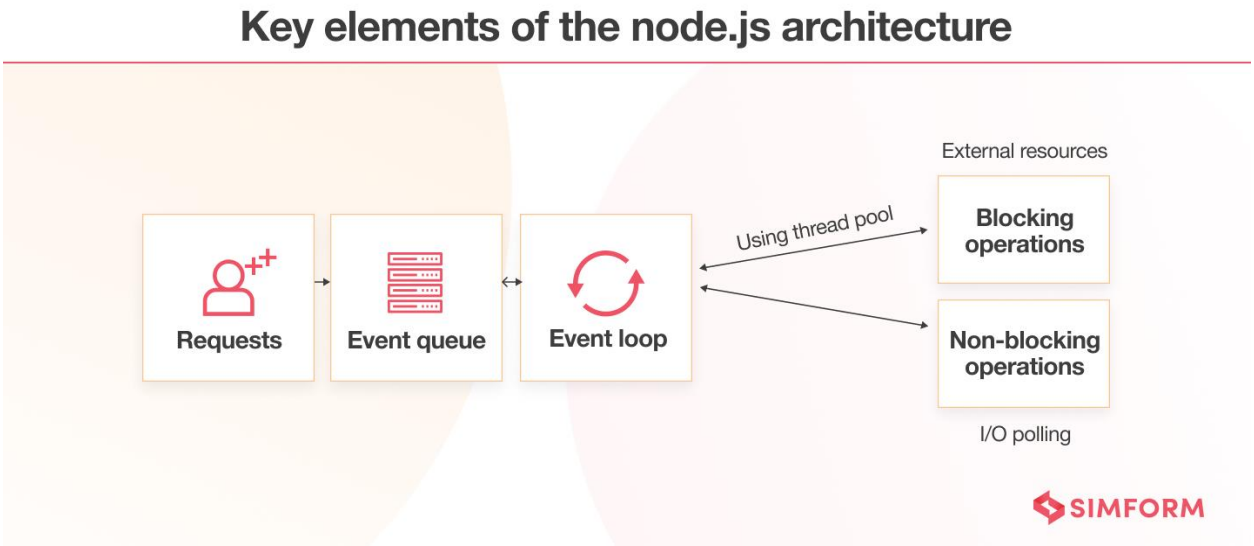
However, In 2018 multithreading was introduced with the worker threads module in Node.js after the 10.5.0 update, Worker threads allow for running several Node.js instances inside a process sharing the same system memory. This solution can help solve some CPU-bound tasks, but it does not make Node.js a multithreading high-performance language.

6. Memory Leaks: If your web application is not coded well, it can be defenseless against memory leaks.

A memory leak is when a piece of code allocates memory but never frees it, resulting in the memory being "leaked." This can eventually lead to the Node.js process running out of memory and crashing.

2. **Nodejs Architecture**

Node.js uses 'Single Threaded Event Loop' architecture to handle multiple concurrent clients. The Node.js processing model is based on a Javascript-event-based model and a callback mechanism. This mechanism based on the event loop allows Node.js to run blocking I/O operations in a non-blocking way. Moreover, scaling is much simpler with a single thread than one thread/new thread per request under ordinary web loads. Now, let's understand the key elements that make up Node.js architecture.



1. Requests – Based on the specific tasks users need to perform in a web application, the requests can either be blocking (complex) or non-blocking (simple).

2. Nodejs server – It accepts user requests, processes them, and returns the results to the corresponding users.

3. Event queue – It stores the incoming requests and passes them sequentially to the Event Pool.

4. Event pool – After receiving the client requests from the event queue, it sends responses to corresponding clients.

5. Thread pool – It contains the threads available for performing those operations necessary to process requests.

6. External resources – The external resources are used for blocking client requests and can be used for computation, data storage, and more.

3. **Nodejs Asynchronous Programming**

Asynchronous programming in Node.js is a key concept that allows you to write non-blocking, efficient code that can handle multiple tasks concurrently without waiting for each task to complete before moving on to the next one. Node.js is designed around an event-driven, non-blocking I/O model, and it uses JavaScript's built-in support for callbacks and Promises to achieve asynchronicity.

Here are some fundamental aspects of asynchronous programming in Node.js:

1. Event Loop: Node.js is built on an event loop that continuously checks for pending tasks and executes them. This loop allows Node.js to handle multiple operations concurrently without blocking the main thread.

2. Callbacks: Callback functions are a common way to handle asynchronous operations in Node.js. Instead of waiting for an operation to complete, you pass a callback function as an argument that will be executed once the operation is done. For example:

```javascript
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
  } else {
    console.log(data);
  }
});
```

3. Promises: Promises provide a more structured way to work with asynchronous code and handle its results or errors. Promises allow you to chain multiple asynchronous operations. The `async/await` syntax is often used with Promises to write cleaner and more readable asynchronous code:

```javascript
const readFileAsync = (fileName) => {
  return new Promise((resolve, reject) => {
    fs.readFile(fileName, 'utf8', (err, data) => {
      if (err) {
        reject(err);
      } else {
        resolve(data);
      }
    });
  });
};

async function readAndLogFile() {
  try {
    const data = await readFileAsync('file.txt');
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}
```

4. Non-blocking I/O: Node.js is optimized for handling I/O operations, such as reading files, making network requests, or interacting with databases, in a non-blocking way. This ensures that your application can continue processing other tasks while waiting for I/O operations to complete.

By embracing asynchronous programming, Node.js can efficiently manage numerous concurrent connections and perform I/O-intensive tasks, making it well-suited for building scalable and high-performance applications, such as web servers and real-time applications.

4. **What is Event ?Explain need of Event loop with neat diagram.**
Node.js is a single-threaded event-driven platform that is capable of running non-blocking, asynchronous programming. These functionalities of Node.js make it memory efficient. The event loop allows Node.js to perform non-blocking I/O operations despite the fact that JavaScript is single-threaded. It is done by assigning operations to the operating system whenever and wherever possible.

Most operating systems are multi-threaded and hence can handle multiple operations executing in the background. When one of these operations is completed, the kernel tells Node.js, and the respective callback assigned to that operation is added to the event queue which will eventually be executed. This will be explained further in detail later in this topic.

Features of Event Loop:

- An event loop is an endless loop, which waits for tasks, executes them, and then sleeps until it receives more tasks.
- The event loop executes tasks from the event queue only when the call stack is empty i.e. there is no ongoing task.
- The event loop allows us to use callbacks and promises.
- The event loop executes the tasks starting from the oldest first.

Example:

```
console.log("This is the first statement");

setTimeout(function(){
   console.log("This is the second statement");
}, 1000);

console.log("This is the third statement");
```

Output:
This is the first statement
This is the third statement
This is the second statement

Explanation: In the above example, the first console log statement is pushed to the call stack, and “This is the first statement” is logged on the console, and the task is popped from the stack. Next, the setTimeout is pushed to the queue and the task is sent to the Operating system and the timer is set for the task. This task is then popped from the stack. Next, the third console log statement is pushed to the call stack, and “This is the third statement” is logged on the console and the task is popped from the stack.

When the timer set by the setTimeout function (in this case 1000 ms) runs out, the callback is sent to the event queue. The event loop on finding the call stack empty takes the task at the top of the event queue and sends it to the call stack. The callback function for the setTimeout function runs the instruction and “This is the second statement” is logged on the console and the task is popped from the stack.

Note: In the above case, if the timeout was set to 0ms then also the statements will be displayed in the same order. This is because although the callback will be immediately

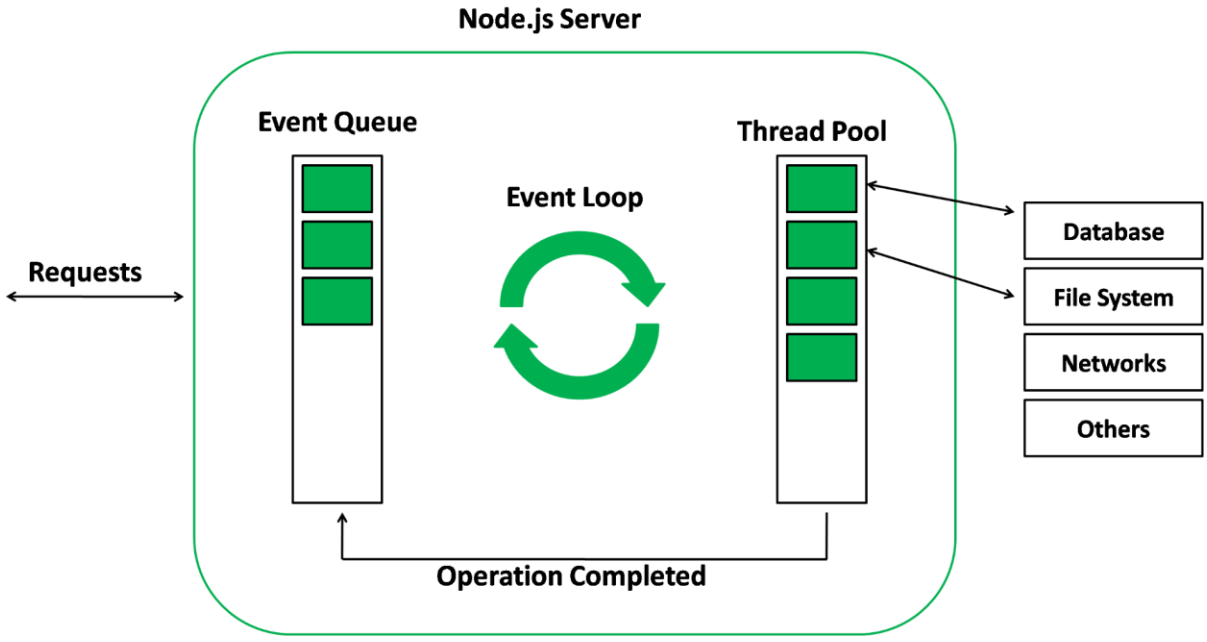
sent to the event queue, the event loop won't send it to the call stack unless the call stack is empty i.e. until the provided input script comes to an end.

Working of the Event loop: When Node.js starts, it initializes the event loop, processes the provided input script which may make async API calls, schedules timers, then begins processing the event loop. In the previous example, the initial input script consisted of console.log() statements and a setTimeout() function which schedules a timer.

When using Node.js, a special library module called libuv is used to perform async operations. This library is also used, together with the back logic of Node, to manage a special thread pool called the libuv thread pool. This thread pool is composed of four threads used to delegate operations that are too heavy for the event loop. I/O operations, Opening and closing connections, setTimeouts are examples of such operations.

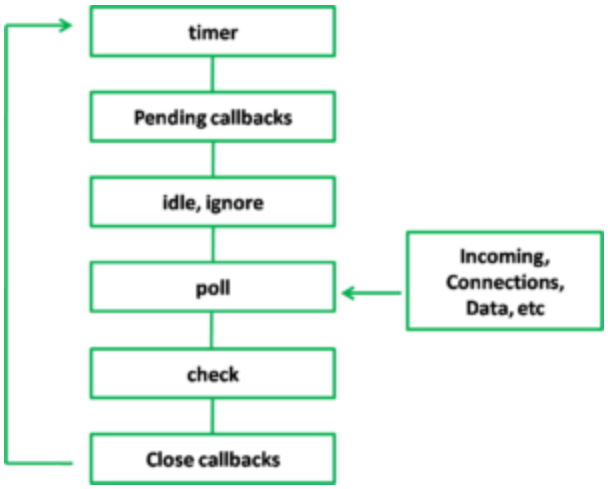
When the thread pool completes a task, a callback function is called which handles the error(if any) or does some other operation. This callback function is sent to the event queue. When the call stack is empty, the event goes through the event queue and sends the callback to the call stack.

The following diagram is a proper representation of the event loop in a Node.js server:



Phases of the Event loop: The event loop in Node.js consists of several phases, each of which performs a specific task. These phases include:

The following diagram shows a simplified overview of the event loop order of operations:



1. Timers: This phase processes timers that have been set using setTimeout() and setInterval().
2. Pending Callbacks: This phase processes any callbacks that have been added to the message queue by asynchronous functions.

3. Idle, Prepare: The “idle.ignore” phase is not a standard phase of the event loop in Node.js. It means it’s Used internally only. The “idle” phase is a period of time during which the event loop has nothing to do and can be used to perform background tasks, such as running garbage collection or checking for low-priority events.

“idle.ignore” is not an official phase of the event loop, it is a way to ignore the idle phase, meaning that it will not use the time of the idle phase to perform background tasks.

4. Poll: This phase is used to check for new I/O events and process any that have been detected.

5. Check This phase processes any setImmediate() callbacks that have been added to the message queue.

6. Close Callbacks: This phase processes any callbacks that have been added to the message queue by the close event of a socket. This means that any code that needs to be executed when a socket is closed is placed in the message queue and processed during this phase.

The event loop is a powerful feature of Node.js that enables it to handle a high number of concurrent connections and perform non-blocking I/O operations. Understanding how the event loop works is essential for building efficient and performant server-side applications in Node.js. With a better understanding of the event loop, developers can take full advantage of the capabilities of Node.js and build high-performance, scalable applications.

5. Asynchronous programming and Callback function in nodejs

Asynchronous Progrramming:
Asynchronous programming in Node.js is a key concept that allows you to write non-blocking, efficient code that can handle multiple tasks concurrently without waiting for each task to complete before moving on to the next one. Node.js is designed around an event-driven, non-blocking I/O model, and it uses JavaScript's built-in support for callbacks and Promises to achieve asynchronicity.

Here are some fundamental aspects of asynchronous programming in Node.js:

1. Event Loop: Node.js is built on an event loop that continuously checks for pending tasks and executes them. This loop allows Node.js to handle multiple operations concurrently without blocking the main thread.

2. Callbacks: Callback functions are a common way to handle asynchronous operations in Node.js. Instead of waiting for an operation to complete, you pass a callback function as an argument that will be executed once the operation is done. For example:

```javascript
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
  } else {
    console.log(data);
  }
});
```

3. Promises: Promises provide a more structured way to work with asynchronous code and handle its results or errors. Promises allow you to chain multiple asynchronous operations. The `async/await` syntax is often used with Promises to write cleaner and more readable asynchronous code:

```javascript
const readFileAsync = (fileName) => {
  return new Promise((resolve, reject) => {
    fs.readFile(fileName, 'utf8', (err, data) => {
      if (err) {
        reject(err);
      } else {
        resolve(data);
      }
    });
  });
```

```
};

async function readAndLogFile() {
  try {
    const data = await readFileAsync('file.txt');
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}
```

4. Non-blocking I/O: Node.js is optimized for handling I/O operations, such as reading files, making network requests, or interacting with databases, in a non-blocking way. This ensures that your application can continue processing other tasks while waiting for I/O operations to complete.

By embracing asynchronous programming, Node.js can efficiently manage numerous concurrent connections and perform I/O-intensive tasks, making it well-suited for building scalable and high-performance applications, such as web servers and real-time applications.

Callback functions:
A callback is a function that is passed as an argument to another function, and is called after the main function has finished its execution. The main function is called with a callback function as its argument, and when the main function is finished, it calls the callback function to provide a result. Callbacks allow you to handle the results of an asynchronous operation in a non-blocking manner, which means that the program can continue to run while the operation is being executed.

Why use Callbacks?

Callbacks are used to handle the results of asynchronous operations in a non-blocking manner. Asynchronous operations are operations that take a significant amount of time to complete, such as network requests, file I/O, and database queries. If these operations were executed synchronously, the program would freeze and wait for the operation to complete before continuing. This can lead to a poor user experience, as the program would appear unresponsive.

Callbacks allow you to continue executing code while the operation is being executed in the background. Once the operation has completed, the callback function is called with the result of the operation. This way, you can ensure that the program remains responsive and the user experience is not impacted.

6. **Explain REPL in nodejs.**

REPL (READ, EVAL, PRINT, LOOP) is a computer environment similar to Shell (Unix/Linux) and command prompt. Node comes with the REPL environment when it is installed. System interacts with the user through outputs of commands/expressions used. It is useful in writing and debugging the codes. The work of REPL can be understood from its full form:

Read : It reads the inputs from users and parses it into JavaScript data structure. It is then stored to memory.
Eval : The parsed JavaScript data structure is evaluated for the results.
Print : The result is printed after the evaluation.
Loop : Loops the input command. To come out of NODE REPL, press ctrl+c twice

Getting Started with REPL:
To start working with REPL environment of NODE; open up the terminal (in case of UNIX/LINUX) or the Command prompt (in case of Windows) and write node and press 'enter' to start the REPL.
The REPL has started and is demarcated by the '>' symbol. Various operations can be performed on the REPL. Below are some of the examples to get familiar with the REPL environment.

7. **What is Module?Explain different types of modules supported by Nodejs. OR**
8. **Nodejs Modules and functions**

In Node.js, Modules are the blocks of encapsulated code that communicate with an external application on the basis of their related functionality. Modules can be a single file or a collection of multiple files/folders. The reason programmers are heavily reliant on modules is because of their reusability as well as the ability to break down a complex piece of code into manageable chunks.

Modules are of three types:

Core Modules
local Modules
Third-party Modules
Core Modules: Node.js has many built-in modules that are part of the platform and come with Node.js installation. These modules can be loaded into the program by using the required function.

Syntax: const module = require('module_name');

The require() function will return a JavaScript type depending on what the particular module returns.

Core Modules	Description
http	creates an HTTP server in Node.js.
assert	set of assertion functions useful for testing.
fs	used to handle file system.
path	includes methods to deal with file paths.
process	provides information and control about the current Node.js process.
os	provides information about the operating system.
querystring	utility used for parsing and formatting URL query strings.
url	module provides utilities for URL resolution and parsing.

Local Modules: Unlike built-in and external modules, local modules are created locally in your Node.js application. Let's create a simple calculating module that calculates various operations. Create a calc.js file that has the following code:

Filename: calc.js

```
exports.add = function (x, y) {
   return x + y;
};

exports.sub = function (x, y) {
   return x - y;
};

exports.mult = function (x, y) {
   return x * y;
};

exports.div = function (x, y) {
   return x / y;
};
```

Since this file provides attributes to the outer world via exports, another file can use its exported functionality using the require() function.

Filename: index.js

```
const calculator = require('./calc');

let x = 50, y = 10;

console.log("Addition of 50 and 10 is "
        + calculator.add(x, y));

console.log("Subtraction of 50 and 10 is "
        + calculator.sub(x, y));

console.log("Multiplication of 50 and 10 is "
        + calculator.mult(x, y));

console.log("Division of 50 and 10 is "
        + calculator.div(x, y));
```

9. **HTTP: Web application using HTTP.**

Steps to create a basic http application in nodeJS:

1. First, make sure you have Node.js installed on your system. You can download it from the official website: https://nodejs.org

2. Create a new directory for your web application and navigate to it in your terminal or command prompt.

3. Create a JavaScript file for your web server. You can name it app.js or something similar.
4. Inside app.js, add the following code to create a basic web server:

```
const http = require('http');

const server = http.createServer((req, res) => {
  // Set the response header with a 200 OK status and plain text content type.
  res.writeHead(200, { 'Content-Type': 'text/plain' });

  // Send a "Hello, World!" message as the response.
  res.end('Hello, World!\n');
});

const port = 3000; // You can change this to any port you prefer.
```

```
server.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}/`);
});
```

10. **NET: Socket Programming**

    Socket programming in Node.js allows you to create network applications that communicate with each other using the WebSocket or TCP/UDP protocols. This enables real-time communication and data exchange between different devices and servers. Below, I'll provide a brief overview of how to perform socket programming in Node.js using the popular `net` and `ws` modules for TCP and WebSocket communication, respectively.

    TCP Socket Programming (using the `net` module):

    1. **Create a TCP Server:**

       ```javascript
       const net = require('net');

       const server = net.createServer((socket) => {
         console.log('Client connected');

         socket.on('data', (data) => {
           console.log(`Received data: ${data.toString()}`);
         });

         socket.on('end', () => {
           console.log('Client disconnected');
         });
       });

       server.listen(3000, () => {
         console.log('Server listening on port 3000');
       });
       ```

    2. **Create a TCP Client:**

       ```javascript
       const net = require('net');

       const client = new net.Socket();

       client.connect(3000, 'localhost', () => {
```

```
  console.log('Connected to server');
  client.write('Hello, server!');
});

client.on('data', (data) => {
  console.log(`Received from server: ${data.toString()}`);
  client.end();
});

client.on('end', () => {
  console.log('Disconnected from server');
});
```

WebSocket Programming (using the `ws` module):

1. **Create a WebSocket Server:**

   ```javascript
   const WebSocket = require('ws');

   const wss = new WebSocket.Server({ port: 8080 });

   wss.on('connection', (ws) => {
     console.log('WebSocket client connected');

     ws.on('message', (message) => {
       console.log(`Received message: ${message}`);
       ws.send('Server received your message: ' + message);
     });
   });
   ```

2. **Create a WebSocket Client:**

   You can create WebSocket clients using browser-based JavaScript, and libraries like `socket.io-client` or `websocket` for Node.js. Here's a basic example using the `websocket` module:

   ```javascript
   const WebSocket = require('websocket').client;

   const client = new WebSocket();

   client.on('connect', (connection) => {
   ```

```
  console.log('Connected to WebSocket server');
  connection.send('Hello, server!');

  connection.on('message', (message) => {
   if (message.type === 'utf8') {
    console.log(`Received from server: ${message.utf8Data}`);
    connection.close();
   }
  });
});

client.connect('ws://localhost:8080/');
```

These examples provide a basic introduction to socket programming in Node.js for both TCP and WebSocket communication. Depending on your specific use case and requirements, you can build more complex and customized real-time applications using these concepts as a foundation. Additionally, you may consider using higher-level libraries like `socket.io` for WebSocket communication, as they provide additional features and abstractions for real-time applications.

11. FS: File System Programming and different functions

The Node.js file system module allows you to work with the file system on your computer.

Common use for the File System module:

Read files The fs.readFile() method is used to read files on your computer.
Example:

```
var http = require('http');
var fs = require('fs');
http.createServer(function (req, res) {
  fs.readFile('demofile1.html', function(err, data) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(data);
    return res.end();
  });
}).listen(8080);
```

Create files:
The File System module has methods for creating new files:

fs.appendFile(): The fs.appendFile() method appends specified content to a file. If the file does not exist, the file will be created:

fs.open(): The fs.open() method takes a "flag" as the second argument, if the flag is "w" for "writing", the specified file is opened for writing. If the file does not exist, an empty file is created:

```
var fs = require('fs');

fs.open('mynewfile2.txt', 'w', function (err, file) {
  if (err) throw err;
  console.log('Saved!');
});
```

fs.writeFile(): The fs.writeFile() method replaces the specified file and content if it exists. If the file does not exist, a new file, containing the specified content, will be created:

Update files: The File System module has methods for updating files:

fs.appendFile(): The fs.appendFile() method appends the specified content at the end of the specified file:

```
var fs = require('fs');

fs.appendFile('mynewfile1.txt', ' This is my text.', function (err) {
  if (err) throw err;
  console.log('Updated!');
});
```

fs.writeFile(): The fs.writeFile() method replaces the specified file and content:

```
var fs = require('fs');

fs.writeFile('mynewfile3.txt', 'This is my text', function (err) {
  if (err) throw err;
  console.log('Replaced!');
});
```

Delete files: To delete a file with the File System module, use the fs.unlink() method.

The fs.unlink() method deletes the specified file:

```
var fs = require('fs');

fs.unlink('mynewfile2.txt', function (err) {
```

```
  if (err) throw err;
  console.log('File deleted!');
});
```

Rename files: To rename a file with the File System module, use the fs.rename() method.

The fs.rename() method renames the specified file:
Ex:

```
var fs = require('fs');

fs.rename('mynewfile1.txt', 'myrenamedfile.txt', function (err) {
  if (err) throw err;
  console.log('File Renamed!');
});
```

12. Buffer and Stream

Buffers and streams are important concepts in Node.js, especially for handling data efficiently, whether reading from a file, sending data over a network, or processing data in real-time. Let's explore both concepts:

Buffer:

A buffer in Node.js is a built-in object used to represent binary data directly in memory. Buffers are essentially arrays of integers that can range from 0 to 255. They are particularly useful for working with binary data, such as reading and writing files or dealing with network protocols. Some key points about buffers include:

- Buffers are instances of the `Buffer` class.
- They have a fixed size, set when you create them.
- Once created, the data in a buffer cannot be resized.
- Buffers can be created from arrays, strings, or even directly from binary data.
- They are commonly used in file I/O operations, network communications, and when working with binary data formats like images or audio files.

Here's a simple example of creating and using a buffer in Node.js:

```javascript
const buffer = Buffer.from('Hello, world!', 'utf-8');
console.log(buffer); // <Buffer 48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 21>
console.log(buffer.toString()); // 'Hello, world!'
```

Streams:

Streams in Node.js are used for reading from or writing to data sources. They are a way to handle data in a more efficient and scalable manner, particularly when dealing with large datasets or real-time data. Node.js includes several built-in stream classes, including `Readable`, `Writable`, and `Transform`. Key characteristics of streams include:

- **Readable Streams**: Used for reading data. They are a source of data that can be read.
- **Writable Streams**: Used for writing data. They are a destination for data to be written.
- **Duplex Streams**: A combination of both readable and writable streams.
- **Transform Streams**: A special type of duplex stream used for data transformation during read and write operations.
- Streams are event-based, allowing you to react to events like `data`, `end`, and `error`.
- Streams are particularly useful when processing data that doesn't fit entirely into memory, as they allow data to be processed in chunks, reducing memory overhead.

Here's a simple example of using a readable and writable stream to copy data from one file to another:

```javascript
const fs = require('fs');

const readStream = fs.createReadStream('source.txt');
const writeStream = fs.createWriteStream('destination.txt');

readStream.pipe(writeStream);

readStream.on('end', () => {
  console.log('Data has been copied.');
});
```

In this example, data is read from `source.txt` using a readable stream and then written to `destination.txt` using a writable stream. The `.pipe()` method is used to efficiently transfer data from the readable stream to the writable stream.

13. Request and Response object methods.

Features:

1. Develops Node.js web applications quickly and easily.

2. It’s simple to set up and personalise.
3. Allows you to define application routes using HTTP methods and URLs.
4. Includes a number of middleware modules that can be used to execute additional requests and responses activities.
5. Simple to interface with a variety of template engines, including Jade, Vash, and EJS.
6. Allows you to specify a middleware for handling errors.

14. Routing in Node Js and Express(Routinhg in nodejs and express goes hand in hand nodejs provides env and through express you are routing the paths.)

Routing In Express

Routing refers to how an application’s endpoints (URIs) respond to client requests. You define routing using methods of the Express app object that correspond to HTTP methods; for example, app.get() to handle GET requests and app.post to handle POST requests. For a full list, see app.METHOD. You can also use app.all() to handle all HTTP methods and app.use() to specify middleware as the callback function.

These routing methods specify a callback function (sometimes called “handler functions”) called when the application receives a request to the specified route (endpoint) and HTTP method. In other words, the application “listens” for requests that match the specified route(s) and method(s), and when it detects a match, it calls the specified callback function.

In fact, the routing methods can have more than one callback function as arguments. With multiple callback functions, it is important to provide next as an argument to the callback function and then call next() within the body of the function to hand off control to the next callback.

The following code is an example of a very basic route.

```
const express = require('express')
const app = express()

// respond with "hello world" when a GET request is made to the homepage
app.get('/', (req, res) => {
  res.send('hello world')
})
```

Route methods

A route method is derived from one of the HTTP methods, and is attached to an instance of the express class.
The following code is an example of routes that are defined for the GET and the POST methods to the root of the app.

```
// GET method route
app.get('/', (req, res) => {
  res.send('GET request to the homepage')
})

// POST method route
app.post('/', (req, res) => {
  res.send('POST request to the homepage')
})
```

Express supports methods that correspond to all HTTP request methods: get, post, and so on. For a full list, see app.METHOD.
There is a special routing method, app.all(), used to load middleware functions at a path for all HTTP request methods. For example, the following handler is executed for requests to the route "/secret" whether using GET, POST, PUT, DELETE, or any other HTTP request method supported in the http module.

```
app.all('/secret', (req, res, next) => {
  console.log('Accessing the secret section ...')
  next() // pass control to the next handler
})
```

- **Route paths**
- Route paths, in combination with a request method, define the endpoints at which requests can be made. Route paths can be strings, string patterns, or regular expressions.

- The characters ?, +, *, and () are subsets of their regular expression counterparts. The hyphen (-) and the dot (.) are interpreted literally by string-based paths.

- If you need to use the dollar character ($) in a path string, enclose it escaped within ([and]). For example, the path string for requests at "/data/$book", would be "/data/([\$])book".
- Here are some examples of route paths based on strings.

This route path will match requests to the root route, /.

```
app.get('/', function (req, res) {
  res.send('root')
})
```

This route path will match requests to /about.

```
app.get('/about', function (req, res) {
  res.send('about')
})
```

This route path will match requests to /random.text.

```
app.get('/random.text', function (req, res) {
  res.send('random.text')
```

})

- **Route parameters**
- Route parameters are named URL segments that are used to capture the values specified at their position in the URL.
- The captured values are populated in the req.params object, with the name of the route parameter specified in the path as their respective keys.

- **Route path:** /users/:userId/books/:bookId
- Request URL: http://localhost:3000/users/34/books/8989
- req.params: { "userId": "34", "bookId": "8989" }

- To define routes with route parameters, simply specify the route parameters in the path of the route as shown below.
- app.get('/users/:userId/books/:bookId', function (req, res) {
- res.send(req.params)
- })
- **Response methods**
- The methods on the response object (res) in the following table can send a response to the client, and terminate the request-response cycle.
- If none of these methods are called from a route handler, the client request will be left hanging.

Method	Description
res.download()	Prompt a file to be downloaded.
res.end()	End the response process.
res.json()	Send a JSON response.
res.jsonp()	Send a JSON response with JSONP support.
res.redirect()	Redirect a request.
res.render()	Render a view template.
res.send()	Send a response of various types.
res.sendFile()	Send a file as an octet stream.
res.sendStatus()	Set the response status code and send its string representation as the response body.

15. Express features,advantages and disadvantages
Features:-

Advantages:-

1. Minimal and Lightweight

Express.js is a minimal and lightweight web framework that provides developers with a basic set of features for building web applications. It is designed to be flexible and

modular, allowing developers to easily add new functionality as needed. Express.js does not include a lot of unnecessary features, making it fast and efficient.

2. Easy to Learn and Use

Express.js is easy to learn and use, making it an ideal choice for both novice and experienced developers. It provides developers with a simple and intuitive API for building web applications. Express.js is built on top of Node.js, which means that developers can leverage the power of the Node.js ecosystem and easily integrate with other Node.js modules.

3. Middleware Support

One of the key features of Express.js is its support for middleware. Middleware is a function that is executed for every HTTP request that is received by the server. Middleware can be used to perform a variety of tasks, such as logging, authentication, and error handling. Express.js provides developers with a wide range of built-in middleware functions, as well as the ability to create custom middleware.

4. Routing

Express.js provides developers with a powerful routing system that makes it easy to handle different HTTP requests for different URLs. Developers can define routes that map to specific URLs and HTTP methods, making it easy to handle different types of requests. Express.js also supports parameterized routes, which allows developers to handle dynamic URLs.

5. Template Engines

Express.js supports a variety of template engines, which makes it easy to render dynamic HTML pages. Developers can choose from a variety of popular template engines, such as EJS, Handlebars, and Pug. Express.js also provides developers with built-in support for serving static files, such as images, CSS, and JavaScript.

6. Database Support

Express.js provides developers with a variety of options for working with databases. It has built-in support for popular databases, such as MongoDB and MySQL, as well as the ability to use other Node.js database modules. Developers can easily connect to databases, perform CRUD operations, and handle database transactions.

7. Scalability

Express.js is designed to be scalable and can handle a large number of concurrent connections. It uses an event-driven, non-blocking I/O model, which allows it to handle multiple requests at the same time. Express.js also supports clustering, which allows developers to scale their applications across multiple CPUs or servers.

Limitations of ExpressJS:

1. Lack of Structure and Convention

One of the primary limitations of using Express.js is that it does not provide a strict structure or convention for organizing your code. While this can be a benefit for some developers who prefer flexibility, it can also lead to messy and hard-to-maintain code.

2. Overhead

Express.js is built on top of Node.js, which means that it can have some overhead in terms of performance. While this overhead is generally small and insignificant for most applications, it can be a concern for applications that require extremely high performance.

3. Limited Built-in Features

While Express.js provides developers with a basic set of features for building web applications, it does not include many advanced features out of the box. Developers will often need to use additional modules or build custom functionality to meet their needs.

4. Steep Learning Curve for Middleware

While middleware is a powerful feature of Express.js, it can have a steep learning curve for developers who are not familiar with the concept. Middleware can be difficult to debug and can introduce unexpected behavior if not used correctly.

5. Lack of Strong Typing

Express.js does not provide strong typing or compile-time checks, which can lead to errors and bugs in your code. This is particularly a concern for larger applications with many dependencies and complex logic.

16. express Routing and cookie management

Cookie management:-

Simple cookie-based session middleware.

A user session can be stored in two main ways with cookies: on the **server** or on the **client**. This module stores the session data on the client within a cookie, while a module like express-session stores only a session identifier on the client within a cookie and stores the session data on the server, typically in a database.

The following points can help you choose which to use:

- cookie-session does not require any database / resources on the server side, though the total session data cannot exceed the browser's max cookie size.
- cookie-session can simplify certain load-balanced scenarios.
- cookie-session can be used to store a "light" session and include an identifier to look up a database-backed secondary store to reduce database lookups.

API

```
var cookieSession = require('cookie-session')
var express = require('express')

var app = express()

app.use(cookieSession({
  name: 'session',
  keys: [/* secret keys */],

  // Cookie Options
  maxAge: 24 * 60 * 60 * 1000 // 24 hours
}))
```

cookieSession(options)

Create a new cookie session middleware with the provided options. This middleware will attach the property session to req, which provides an object representing the loaded session. This session is either a new session if no valid session was provided in the request, or a loaded session from the request.

The middleware will automatically add a Set-Cookie header to the response if the contents of req.session were altered. ***Note*** that no Set-Cookie header will be in the response (and thus no session created for a specific user) unless there are contents in the session, so be sure to add something to req.session as soon as you have identifying information to store for the session.

Options

Cookie session accepts these properties in the options object.

name

The name of the cookie to set, defaults to session.

keys

The list of keys to use to sign & verify cookie values, or a configured Keygrip instance. Set cookies are always signed with keys[0], while the other keys are valid for verification,

allowing for key rotation. If a Keygrip instance is provided, it can be used to change signature parameters like the algorithm of the signature.

secret

A string which will be used as a single key if keys are not provided.

Cookie Options

Other options are passed to cookies.get() and cookies.set() allowing you to control security, domain, path, and signing among other settings.

The options can also contain any of the following (for the full list, see cookies module documentation:

- maxAge: a number representing the milliseconds from Date.now() for expiry
- expires: a Date object indicating the cookie's expiration date (expires at the end of session by default).
- path: a string indicating the path of the cookie (/ by default).
- domain: a string indicating the domain of the cookie (no default).
- sameSite: a boolean or string indicating whether the cookie is a "same site" cookie (false by default). This can be set to 'strict', 'lax', 'none', or true (which maps to 'strict').
- secure: a boolean indicating whether the cookie is only to be sent over HTTPS (false by default for HTTP, true by default for HTTPS). If this is set to true and Node.js is not directly over a TLS connection, be sure to read how to setup Express behind proxies or the cookie may not ever set correctly.
- httpOnly: a boolean indicating whether the cookie is only to be sent over HTTP(S), and not made available to client JavaScript (true by default).
- signed: a boolean indicating whether the cookie is to be signed (true by default).
- overwrite: a boolean indicating whether to overwrite previously set cookies of the same name (true by default).

req.session

Represents the session for the given request.

.isChanged

Is true if the session has been changed during the request.

.isNew

Is true if the session is new.

.isPopulated

Determine if the session has been populated with data or is empty.

req.sessionOptions

Represents the session options for the current request. These options are a shallow clone of what was provided at middleware construction and can be altered to change cookie setting behavior on a per-request basis.

Destroying a session

To destroy a session simply set it to null:

req.session = null

Saving a session

Since the entire contents of the session is kept in a client-side cookie, the session is “saved” by writing a cookie out in a Set-Cookie response header. This is done automatically if there has been a change made to the session when the Node.js response headers are being written to the client and the session was not destroyed.

17. Express different types of of methods of request and response objects

Index	Properties	Description
1.	req.app	This is used to hold a reference to the instance of the express application that is using the middleware.
2.	req.baseurl	It specifies the URL path on which a router instance was mounted.
3.	req.body	It contains key-value pairs of data submitted in the request body. By default, it is undefined, and is populated when you use body-parsing middleware such as body-parser.
4.	req.cookies	When we use cookie-parser middleware, this property is an object that contains cookies sent by the request.
5.	req.fresh	It specifies that the request is "fresh." it is the opposite of req.stale.
6.	req.hostname	It contains the hostname from the "host" http header.
7.	req.ip	It specifies the remote IP address of the request.
8.	req.ips	When the trust proxy setting is true, this property contains an array of IP addresses specified in the ?x-forwarded-for? request header.
9.	req.originalurl	This property is much like req.url; however, it retains the original request URL, allowing you to rewrite req.url freely for internal routing purposes.
10.	req.params	An object containing properties mapped to the named route ?parameters?. For example, if you have the route /user/:name, then the "name" property is available as req.params.name. This object defaults to {}.
11.	req.path	It contains the path part of the request URL.
12.	req.protocol	The request protocol string, "http" or "https" when requested with TLS.
13.	req.query	An object containing a property for each query string parameter in the route.
14.	req.route	The currently-matched route, a string.
15.	req.secure	A Boolean that is true if a TLS connection is established.
16.	req.signedcookies	When using cookie-parser middleware, this property contains signed cookies sent by the request, unsigned and ready for use.
17.	req.stale	It indicates whether the request is "stale," and is the opposite of req.fresh.
18.	req.subdomains	It represents an array of subdomains in the domain name of the request.
19.	req.xhr	A Boolean value that is true if the request's "x-requested-with" header field is "xmlhttprequest", indicating that the request was issued by a client library such as jQuery

Request Object Method	Description
`req.params`	Retrieves route parameters.
`req.query`	Retrieves query parameters from the URL.
`req.body`	Retrieves the request body data, typically for POST or PUT requests.
`req.headers`	Retrieves the HTTP headers of the request.
`req.cookies`	Retrieves cookies from the request.
`req.get(header)`	Retrieves the value of a specific HTTP header.
`req.is(type)`	Checks if the request content type matches a given type.
`req.ip`	Retrieves the client's IP address.
`req.path`	Retrieves the path part of the URL.
`req.originalUrl`	Retrieves the original URL requested by the client.

Response Object Method	Description
`res.send(data)`	Sends a response with the provided data as the response body.
`res.status(code)`	Sets the HTTP status code of the response.
`res.json(data)`	Sends a JSON response.
`res.redirect([status,] path)`	Redirects the client to another URL.
`res.cookie(name, value[, options])`	Sets a cookie in the response.
`res.clearCookie(name[, options])`	Clears a cookie in the response.
`res.header(name, value)`	Sets an HTTP response header.
`res.type(type)`	Sets the content type of the response.
`res.sendStatus(statusCode)`	Sends an empty response with the specified status code.

Rege

18. Nodejs and Express simple program to display message

Hello world example

```
const express = require('express')
const app = express()
```

```
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

19. **Program based on Routing resolved your comment bruv (Already done above)**
20. **Program to send file or redirect to another web page**

HTML

```
<!DOCTYPE html>
<html>
<head>
    <title>Sample HTML Page</title>
</head>
<body>
    <h1>Hello, this is a sample HTML page!</h1>
</body>
</html>
```

Express

```
const express = require('express');
const app = express();
const path = require('path');

// Define a route to serve the HTML file
app.get('/sample', (req, res) => {
    const filePath = path.join(__dirname, 'sample.html'); // Path to your HTML file
    res.sendFile(filePath);
});

// Start the Express server
const port = 3000;
app.listen(port, () => {
    console.log(`Server is running on port ${port}`);
});
```

Java Script Programs:

Event handling

```
<!DOCTYPE html>
<html>
<head>
  <title>Event Handling Example</title>
</head>
<body>
  <button id="myButton">Click Me</button>
  <p id="output">Click the button to see a message.</p>

  <script>
    // Get references to the button and output elements
    const button = document.getElementById('myButton');
    const output = document.getElementById('output');

    // Define an event handler function
    function handleClick() {
      output.textContent = 'Button clicked!'; }

    // Add an event listener to the button element
    button.addEventListener('click', handleClick);
  </script>
</body>
</html>
```

Program based on DOM(Form Handling)

```
<!DOCTYPE html>
<html>
<head>
  <title>Form Handling Example</title>
</head>
<body>
  <h1>Form Handling Example</h1>
  <form id="myForm">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name" required><br><br>

    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required><br><br>

    <label for="message">Message:</label>
    <textarea id="message" name="message" required></textarea><br><br>

    <button type="submit">Submit</button>
  </form>
```

```
  <div id="output"></div>

  <script>
    // Get a reference to the form and the output element
    const form = document.getElementById('myForm');
    const output = document.getElementById('output');

    // Define the form submission event handler
    form.addEventListener('submit', function(event) {
      event.preventDefault(); // Prevent the default form submission behavior

      // Get form input values
      const name = form.name.value;
      const email = form.email.value;
      const message = form.message.value;

      // Display the submitted data
      output.innerHTML = `
        <h2>Submitted Data:</h2>
        <p><strong>Name:</strong> ${name}</p>
        <p><strong>Email:</strong> ${email}</p>
        <p><strong>Message:</strong> ${message}</p>
      `;
    });
  </script>
</body>
</html>
```

Validation

```
<!DOCTYPE html>
<html>
  <head>
    <title>Form Validation</title>
  </head>
  <body>
    <h1>Form Validation with JavaScript</h1>
    <form id="myForm" onsubmit="return validateForm()">
      <label for="name">Name:</label>
      <input type="text" id="name" placeholder="Enter your name" /><br /><br />
      <label for="email">Email:</label>
      <input
        type="text"
        id="email"
        placeholder="Enter your email"
      /><br /><br />
```

```
    <label for="password">Password:</label>
    <input
      type="password"
      id="password"
      placeholder="Enter your password"
    /><br /><br />
    <button type="submit">Submit</button>
  </form>
  <p id="errorText" style="color: red"></p>
  <script>
    function validateForm() {
      // Get form elements
      var name = document.getElementById("name").value;
      var email = document.getElementById("email").value;
      var password = document.getElementById("password").value;
      var errorText = document.getElementById("errorText");
      // Simple validation: check if fields are empty
      if (name === "" || email === "" || password === "") {
        errorText.textContent = "All fields are required";
        return false; // Prevent form submission
      }
      // Validate email using a simple regular expression
      var emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
      if (!emailPattern.test(email)) {
        errorText.textContent = "Invalid email format";
        return false; // Prevent form submission
      }
      // Password length validation
      if (password.length < 6) {
        errorText.textContent = "Password must be at least 6 characters long";
        return false; // Prevent form submission
      }
      // If all validation checks pass, the form is submitted
      errorText.textContent = ""; // Clear any previous error message
      return true;
    }
  </script>
 </body>
</html>
```

Call back function

```
// A function that simulates an asynchronous operation
function fetchData(callback) {
 setTimeout(function() {
  const data = "This is the fetched data.";
  callback(data);
```

```
  }, 2000);
}

// Define a callback function to handle the fetched data
function processFetchedData(data) {
  console.log("Data received:", data);
}

// Call the fetchData function and provide the callback function
fetchData(processFetchedData);

console.log("Fetching data..."); // This will be displayed before the data is fetched
```

Arrow function

```
    var square = (x) => {
        return (x*x);
    };
    console.log(square(9));
```

Program based on Promise

```
// A function that returns a Promise that resolves after a delay
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const data = "This is the fetched data.";
      resolve(data);
    }, 2000);
  });
}

// Using the Promise
console.log("Fetching data...");

fetchData()
  .then((data) => {
    console.log("Data received:", data);
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```