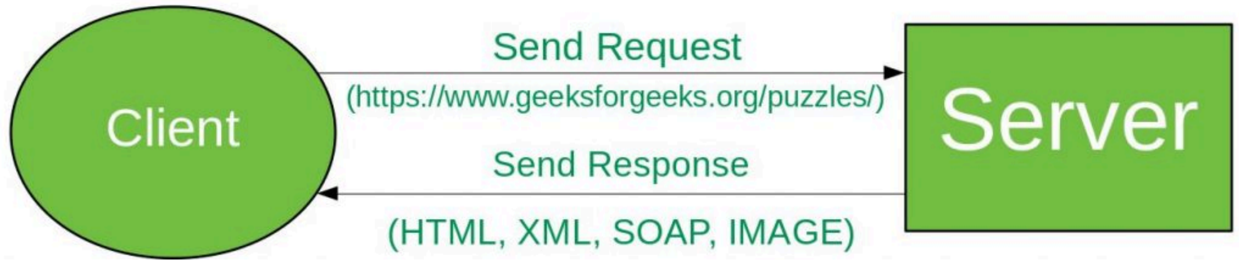
INTERNET PROGRAMMING

Chapter 1:

1. **What is REST api ? What are the principles of REST api. (X3)**

- **RE**presentational **S**tate **T**ransfer (REST) is an architectural style that defines a set of constraints to be used for creating web services. It is an architectural style for creating websites using the HTTP protocol.
- A REST API is an application programming interface that adheres to the constraints of REST architectural style & enables interaction with RESTful web services.
- A web service is a set of open protocols & standards used for exchanging data between client-server applications. Web services that follow the REST architecture are known as RESTful web services.



Post, Get, Put, Patch, Delete - CRU(modify)D

Principles of REST API:

1. Client-Server decoupling: In a REST API design, client & server programs must be independent. The client software should only know the URI of the requested resource; it should have no additional interaction with the server application.

2. Uniform Interface: All API queries for the same resource should look the same regardless of where they come from. The REST API should ensure that similar data, such as a user's name or email address, is assigned to just one uniform resource identifier (URI).

3. Statelessness: REST APIs are stateless, meaning each request must contain all the information needed to process it.

4. Layered System architecture: REST API requests & responses are routed through many tiers. REST APIs must be designed so neither the client nor the server can tell whether they communicate with the final application or an intermediary.

5. Cacheable: Wherever feasible, resources should be cacheable on the client or server side. Server responses must additionally indicate if caching is authorized for the offered assistance. The objective is to boost client-side speed while enhancing server-side scalability.

6. Code on Demand: REST APIs typically provide static resources, but in rare cases, responses may include executable code (such as Java applets). In these cases, perform the code when necessary.

REST API Communication:

- REST APIs communicate through HTTP requests, performing st&ard database functions (CRUD) within a resource.

- HTTP methods like GET, POST, PUT, & DELETE are employed.

- The design resembles a website in a web browser, utilizing built-in HTTP functionality.

- Resource information can be delivered in various formats, including JSON, HTML, XML, Python, PHP, or plain text.

CODE: Dynamic List Creation (JavaScript):

```
 Dynamic List Creation (JavaScript):

// Client-side code using fetch

fetch('https://api.example.com/users')

  .then(response => response.json())

  .then(data => console.log(data));

// Server-side code using Node.js & Express

const express = require('express');

const app = express();

app.get('/users', (req, res) => {

  // Handle the request & send back data

  res.json({ users: [...] });

});

app.listen(3000, () => {

  console.log('Server is running on port 3000');
});
```

2. Differentiate between JSON & XML.

Ans.

Based on	**JSON**	**XML**
St&s for	*JSON* means JavaScript Object Notation.	*XML* means Extensible Markup Language.

Example:	{"Geeks":[{ "firstName":"Vivek", "lastName":"Kothari" }, { "firstName":"Suraj", "lastName":"Kumar" }, { "firstName":"John", "lastName":"Smith" }, { "firstName":"Peter", "lastName":"Gregory" }]}	<Geeks> <Geek> <firstName>Vivek</firstName> <lastName>Kothari</lastName> </Geek> <Geek> <firstName>Suraj</firstName> <lastName>Kumar</lastName> </Geek> <Geek> <firstName>John</firstName> <lastName>Smith</lastName> </Geek> <Geek> <firstName>Peter</firstName> <lastName>Gregory</lastName> </Geek> </Geeks>
Format	JSON uses a maplike structure with key-value pairs.	XML stores data in a tree structure with namespaces for different data categories.

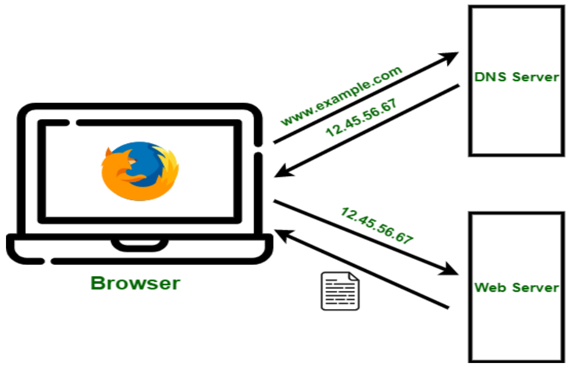
Syntax	The syntax of JSON is more compact & easier to read & write.	The syntax of XML substitutes some characters for entity references, making it more verbose.
Parsing	You can parse JSON with a st&ard JavaScript function.	You need to parse XML with an XML parser.
Schema documentation	JSON is simple & more flexible.	XML is complex & less flexible.
Data types	JSON supports numbers, objects, strings, & Boolean arrays.	XML supports all JSON data types & additional types like Boolean, dates, images, & namespaces.
Ease of use	JSON has smaller file sizes & faster data transmission.	XML tag structure is more complex to write & read & results in bulky files.
Security	JSON is safer than XML.	You should turn off DTD when working with XML to mitigate potential security risks.

3. What is DNS? Explain the working of DNS.

Ans. DNS (Domain Name System) is a hierarchical & decentralized naming system for Internet connected resources. DNS maintains a list of domain names along with the resources, such as IP addresses, that are associated with them.

The most prominent function of DNS is the translation of human-friendly domain names (such as mozilla.org) to a numeric IP address (such as 192.0.2.172); this process of mapping a domain name to the appropriate IP address is known as a **DNS lookup.** By contrast, a reverse DNS lookup (rDNS) is used to determine the domain name associated with an IP address.

An application called a DNS resolver is in charge of translating domain names into IP addresses. The DNS resolver contacts a DNS server to seek the IP address associated with a domain name when a user types it into their web browser.



DNS Server: A DNS server is a piece of software that keeps track of domain names & IP addresses. It answers with the appropriate IP address to requests from DNS resolvers.

The initial point of contact in the DNS system is a DNS root server. It offers details on the DNS servers in charge of top-level domains (TLDs), including.com,.org, &.net.

TLD Server: A TLD server is a DNS server that is in charge of keeping track of data on domain names that fall under a certain top-level domain, such as.com or.org.

Working of dns:

The following are the steps in a DNS Lookup:

1. At the first time a user enters URL as www.msn.cominto the address bar of a veb browser or use age immediately the internal query flows over the Internet and it is then received by a DNS recursive resolver.

2. After receiving the URL, the recursive resolver sends a query to a DNS root name server.

3.Then the root name server gets responded by the resolver with the new address which is a Top-level Domain Then the server (for example org or com), which is used to store the information about is resethne domains. While searching for msn.com, the request is pointed towards the org or com Top-Level Dom _n.

4. The request to the org or .com Top-Level Domain is being made by the resolver.

5. Top Level Domain server then sends response using the logical IP address of the domain's name server fe

example, msn.com.

6. Finally, the recursive resolver fires a query to the respective domain's name server.

7. The logical IP address (e.g. 198.168.100.201) for msn.com is then returned back to the resolver which is from the name server.

8. The Domain Name System resolver which then sends response to the web browser or user agent with the help of

IP address from which the request received from the domain. The browser or user agent will make a request for the web page when these above 8 steps of the DNS lookup returned the IP address for msn.com:

9. The browser or user agent sends a proper HTT request to the internal IP address.

10. The server which is located at that IP address then returns the webpage (only contents and not a file) which then gets loaded on the browser.

A records – points to IPv4 address of machine where website is hosted

AAAA records – points to IPv6 address of machine where website is hosted

MX – points to email servers

CNAME – canonical name for alias points hostname to hostname

ANAME – Auto resolved alias, works like cname but points hostname to IP of hostname

NS – nameservers for subdomains

PTR – IP address to hostname

SOA – containing administrative information about the DNS zone

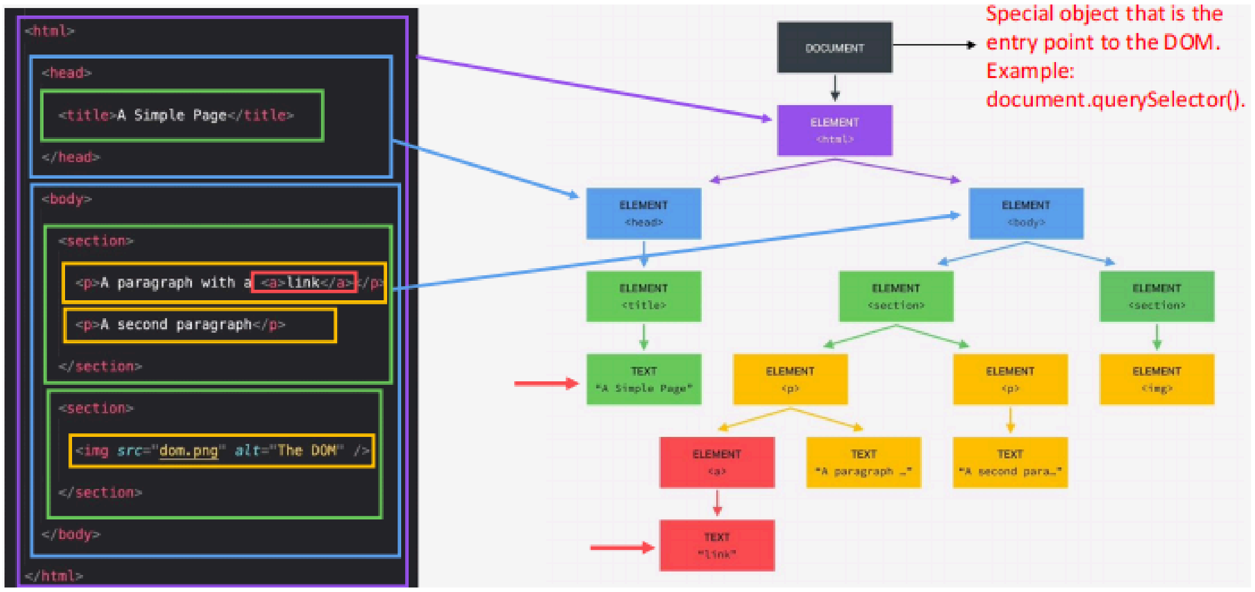
SRV – service record for other services

TXT – Text records mostly used for verification, SPF, DKIM, DMARC & more

CAA – certificate authority record for SSL/TLS certificate

4. Short Note on DOM.

Ans. 1. Definition of DOM: The Document Object Model (DOM) is the data representation of the objects that comprise the structure and content of a document on the web. This is a "tree structure" representation created by the browser that enables the HTML structure to be easily accessed by programming languages — for example the browser itself uses it to apply styling & other information to the correct elements as it renders a page, & developers can manipulate the DOM with JavaScript after the page has been rendered.



```html
<html>
  <head>
    <title>A Simple Page</title>
  </head>
  <body>
    <section>
      <p>A paragraph with a <a>link</a></p>
      <p>A second paragraph</p>
    </section>
    <section>
      <img src="dom.png" alt="The DOM" />
    </section>
  </body>
</html>
```

Special object that is the entry point to the DOM. Example: document.querySelector().

Methods of Document Object:

- write("string"): Writes the given string on the document.

- getElementById(): Returns the element with the given id.

- getElementsByName(): Returns elements with the given name.

- getElementsByTagName(): Returns elements with the given tag name.

- getElementsByClassName(): Returns elements with the given class name.

Types of DOM Supported by JavaScript:

1. Legacy DOM:

- Early JavaScript versions used this model.
- Provides read-only properties (e.g., title, URL).
- Includes lastModified information about the document.
- Methods to set and get document property values.

Document Properties of Legacy DOM:

- alinkcolor: Color of activated links.
- vlinkcolor: Color of visited links.
- linkcolor: Color of unvisited links.
- Title: Contents of the title tag.
- Fgcolor: Default text color of the document.
- Bgcolor: Background color of the document.

2. W3C DOM:

- Empowers JavaScript to create dynamic HTML.

- Allows modification of HTML elements, attributes, and CSS styles.
- Enables addition, removal, and rearrangement of HTML elements.
- Facilitates reaction to HTML events and creation of new events.
- Interaction with the DOM API to change or modify an HTML document.

CODE:Adding New Elements to the DOM, Getting or Setting HTML Contents, Inserting HTML Content, Removing and Replacing Elements, Querying Elements and Setting Styles

```
<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <meta name="viewport" content="width=device-width, initial-scale=1.0">

  <title>DOM Manipulation Example</title>

  <style>

    #main {

      border: 1px solid #ccc;

      padding: 10px;

      margin: 10px;

    }

    button {

      margin-top: 10px;

    }

  </style>
```

```
</head>
<body>
  <div id="main">
    <h1 id="title">Hello World!</h1>
    <p id="hint">This is a simple paragraph.</p>
  </div>
  <button onclick="manipulateDOM()">Manipulate DOM</button>
  <script>
    function manipulateDOM() {
      var newDiv = document.createElement("div");
      var newContent = document.createTextNode("Hi, how are you doing?");
      newDiv.appendChild(newContent);
      document.body.appendChild(newDiv);
      var contents = document.getElementById("main").innerHTML;
      alert(contents);
      var mainDiv = document.getElementById("main");
      mainDiv.insertAdjacentHTML('beforebegin', '<p>This is paragraph
one.</p>');
      mainDiv.insertAdjacentHTML('afterbegin', '<p>This is paragraph
two.</p>');
      mainDiv.insertAdjacentHTML('beforeend', '<p>This is paragraph
three.</p>');
      mainDiv.insertAdjacentHTML('afterend', '<p>This is paragraph
four.</p>');
      var parentElem = document.getElementById("main");
      var childElem = document.getElementById("hint");

      parentElem.removeChild(childElem);

      var newParagraph = document.createElement("p");

      var newText = document.createTextNode("This is a new paragraph.");

      newParagraph.appendChild(newText);

      var oldParagraph = document.getElementById("title");

      parentElem.replaceChild(newParagraph, oldParagraph);

      document.querySelector("p").style.backgroundColor = "lightblue";

    }

  </script>

</body>

</html>
```

5. Short Note on DOM Manipulation.

Ans. DOM: The Document Object Model (DOM) is the data representation of the objects that comprise the structure and content of a document on the web. This is a "tree structure" representation created by the browser that enables the HTML structure to be easily accessed by programming languages.

DOM Manipulation: DOM manipulation refers to the dynamic modification of HTML & CSS content on a webpage using JavaScript. It allows developers to interact with the Document Object Model, adding, modifying, or deleting elements to create a responsive & dynamic user interface.

Key Concepts:

1. Selecting Elements:

- Use methods like `getElementById`, `getElementsByClassName`, `getElementsByTagName`, or `querySelector` to select HTML elements.

- The selected element is then stored in a variable for further manipulation.

```
JavaScript

// Example: Selecting an element by ID

let myElement = document.getElementById('myId');
```

2. Modifying Content:

- Update the content of selected elements using properties like `innerHTML` or `textContent`.

- `innerHTML` is often used for HTML content, while `textContent` is used for plain text.

```
JavaScript

// Example: Modifying inner HTML content

  myElement.innerHTML = 'New Content';
```

3. Changing Attributes:

- Adjust attributes such as `src`, `href`, or `class` to dynamically update element properties.

- This allows for dynamic changes in images, links, or styling.

```javascript

```
// Example: Changing the source attribute of an image

let myImage = document.getElementById('myImage');

myImage.src = 'new_image.jpg';
~~~

4. Creating New Elements:

- Generate new HTML elements using `createElement` & append them to the document.

- This is useful for dynamically adding content to the page.

```javascript

~~~
// Example: Creating & appending a new paragraph

let newParagraph = document.createElement('p');

newParagraph.textContent = 'This is a new paragraph.';

document.body.appendChild(newParagraph);
~~~

5. Event Handling:

- Attach event listeners to respond to user actions like clicks or key presses.

- The event listener function specifies what should happen when the event occurs.

```javascript

~~~
// Example: Adding a click event listener
```
```

```
myElement.addEventListener('click', function() {

    alert('Element Clicked!');

});
```

6. Style Manipulation:

   - Dynamically change the style of elements using the `style` property.

   - This provides flexibility in adjusting visual aspects dynamically.

   ```javascript

```
// Example: Changing the background color dynamically

myElement.style.backgroundColor = 'blue';
```

Benefits of DOM Manipulation:

- Dynamic User Interfaces: Enables the creation of dynamic & responsive user interfaces.

- Real-time Updates: Allows real-time updates to content without the need for page reloads.

- User Interaction: Facilitates interaction with user input, enhancing the overall user experience.

```
Example: Dynamic List Creation:

HTML:

<ul id="myList"></ul>

<button onclick="addListItem()">Add List Item</button>

JavaScript:
```
```

```
function addListItem() {

    // Select the unordered list

    let myList = document.getElementById('myList');

    // Create a new list item

    let newItem = document.createElement('li');

    newItem.textContent = 'New Item';

    // Append the new item to the list

    myList.appendChild(newItem);

}
```

6. Short Note on JSX.

Ans. JSX (JavaScript XML): JSX, or JavaScript XML, is a syntax extension for JavaScript, primarily associated with React for building user interfaces. It enables the incorporation of HTML-like tags within JavaScript code, allowing a more declarative & intuitive way to describe the structure of UI components.

Key Features:

1. HTML-Like Syntax: JSX syntax closely resembles HTML, making it easy for developers to write & understand the structure of React components. This similarity simplifies the process of transitioning from regular HTML to JSX.

2. Expression Embedding: JSX supports embedding JavaScript expressions within curly braces `{}`. This feature allows dynamic content & variables to be seamlessly

integrated into the JSX code. For example, you can embed variables, function calls, or any valid JavaScript expression.

Example: JSX in React Component

```
import React from 'react';

// Example React Component using JSX

const Greeting = (props) => {

  return (

    <div>

      <h1>Hello, {props.name}!</h1>

      <p>Welcome to JSX example.</p>

    </div>

  );

};

// Usage of the Greeting component

const App = () => {

  return <Greeting name="User" />;

};

export default App;
```

Explanation: In the example, the `Greeting` component is defined using JSX syntax. The component includes HTML-like tags such as `<div>`, `<h1>`, & `<p>`.

The dynamic content, `props.name`, is embedded using curly braces within the JSX code.

Benefits of JSX:

1. Readability: JSX enhances code readability by providing a structure that closely resembles HTML. This similarity makes it easier for developers to understand & visualize the component hierarchy.

2. Component Composition: JSX simplifies the composition of complex UI components. Components can be nested & reused, contributing to a modular & maintainable codebase.

3. Integration with JavaScript: JSX seamlessly integrates with JavaScript expressions. This integration allows developers to use JavaScript features within JSX, making it a versatile & expressive syntax for building dynamic UIs.

7. Short Note on URL.

Ans. A URL (Uniform Resource Locator) serves as the address for uniquely identifying & locating resources on the World Wide Web.

In case of computer network, to retrieve resource documents or web pages, a URL is used. URL (Uniform Resource Locator) is a global address.

• URLs is an address or link to give a request to a server for fetching the contents of a web page using HTTP.URL can also be used for accessing data from a database using Java DataBaseConnectivity, accessing email using mail to function.

• A URI means Uniform Resource Identifier. It is a set of characters that can be used to identify a physical or logical address. URI has syntax rules for ensuring uniformity.

• URI maintains extensibility with the help of a hierarchical naming scheme.

URI, URL, URN

• URL: URL specifies a location on the network. It is a technique for retrieving the contents.

• URN: It specifies the URN scheme.

Syntax of URL

http://www.abcd.co.in/Folder_Name/index.htm

From the above URL, it is shown that HTTP protocol is being used and consists of following parts:

• http://www.abcd.co.in/: It is the domain name which is known as server port id or the web host.

• /Folder-Name/: It represents a web app name. It also indicates that the website page is stored in a web server. For example, Tomcat Server acts like a middleware server.

• index.html: It is a web page file name. The "htm or html" is an extension for the HTML file.

2. Examples:

- Consider the following examples of URLs:

  - `https://tsec.edu/`: Points to the main page of a website.

  - `https://tsec.edu/it-about-department/it-time-table/`: Specifies a more specific page within the website.

  - `https://tsec.edu/wp-content/uploads/2021/07/S1_IT.pdf`: Direct link to a PDF file.

10. Code Example (Javascript):

```
const myURL = new URL('https://example.com/path?query=value#fragment');

console.log(myURL.hostname);  // Output: example.com

console.log(myURL.search);    // Output: ?query=value
```

11. Usage in Web Development:

 - URLs play a vital role in web development, serving as links, facilitating routing, & providing a structured means to access content on the internet.

8. Short Note on NPM.

Ans. 1. Definition: NPM stands for Node Package Manager, serving as a library & registry for JavaScript software packages. It includes command-line tools for package installation & dependency management.

2. Key Points:

 - NPM is an online repository for publishing open-source Node.js projects.

- It functions as a command-line utility, facilitating interactions with the repository, aiding in package installation, & managing versions & dependencies.

3. Significance: NPM is a central hub for JavaScript code sharing, contributing to the collaborative nature of the JavaScript community..

4. Dependency Management:

   - While manual management of project dependencies is possible, it becomes challenging as projects expand. NPM resolves this issue by handling dependency & package management efficiently.

   - The `package.json` file in your project contains a comprehensive list of dependencies. Running `npm install` ensures the installation of all project dependencies, streamlining the development process.

   - Version specifications in `package.json` prevent updates from breaking the project, providing stability.

5. NPM Programs: The `package.json` file includes a `scripts` attribute, enabling the definition of custom scripts for various tasks:

   - `npm test`: Executes tests for your project.

   - `npm build`: Initiates the project build process.

   - `npm start`: Runs the project locally.

6. Code Example: - `package.json` file:

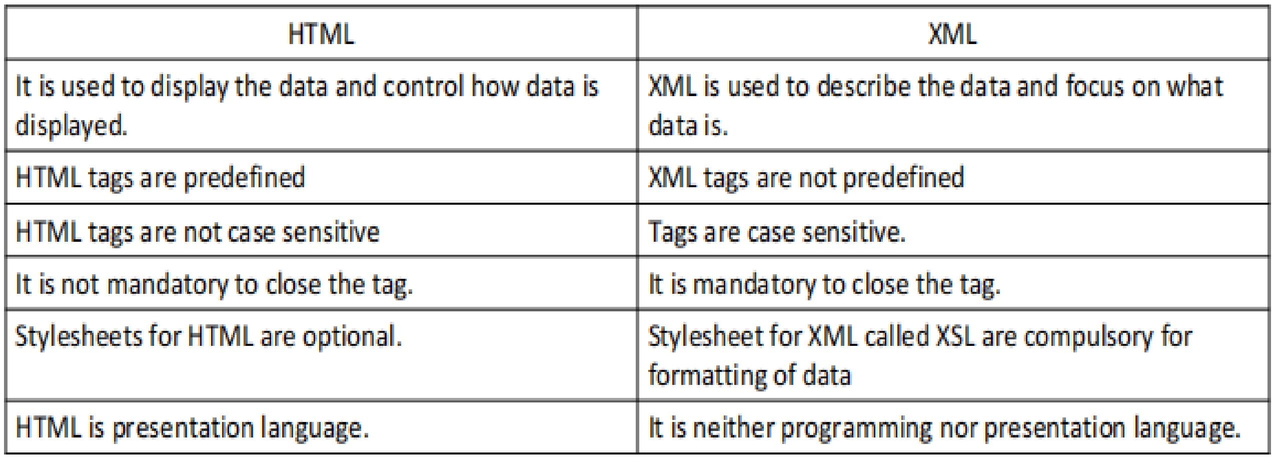
```
{
  "name": "my-project",
  "version": "1.0.0",
  "description": "A brief description of my project",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "your-build-comm&",
    "start": "your-start-comm&"
  },
  "dependencies": {
    "your-dependency": "^1.0.0"
  },
  "devDependencies": {},
  "keywords": [],
  "author": "Your Name",
  "license": "MIT"
}
```

- Running scripts:

```
npm test

npm build

npm start
```

9. Difference between HTML & XML.

Ans.



HTML	XML
It is used to display the data and control how data is displayed.	XML is used to describe the data and focus on what data is.
HTML tags are predefined	XML tags are not predefined
HTML tags are not case sensitive	Tags are case sensitive.
It is not mandatory to close the tag.	It is mandatory to close the tag.
Stylesheets for HTML are optional.	Stylesheet for XML called XSL are compulsory for formatting of data
HTML is presentation language.	It is neither programming nor presentation language.

XML Code: <!-- book.xml -->

```
<book>

  <title>Introduction to XML</title>

  <author>John Doe</author>

  <price>29.99</price>
```

```
</book>

HTML Code:<!-- book.html -->

<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <meta name="viewport" content="width=device-width, initial-scale=1.0">

  <title>Book Details</title>

</head>

<body>

  <h1>Book Information</h1>

  <p><strong>Title:</strong> Introduction to XML</p>

  <p><strong>Author:</strong> John Doe</p>

  <p><strong>Price:</strong> $29.99</p>

</body>

</html>
```

10. Explain XML Schema.

Ans. XML Schema:

XML Schema Definition (XSD) is a way to describe & validate the structure & data types of XML documents. It provides a set of rules & constraints to ensure the integrity & validity of XML documents.

- XML Schema is a specification that defines the elements, attributes, data types, & structure of XML documents.
- It allows you to define the structure of XML documents & validate whether an XML instance document adheres to that structure.
- XML Schema supports the use of namespaces, allowing you to create modular & reusable schema components.
- It includes built-in data types (e.g., string, integer, date) & allows you to define custom data types.
- You can declare elements, specifying their name, data type, occurrence (e.g., mandatory or optional), & other constraints.
- Attributes of elements can be declared with their name, data type, & other constraints.
- It supports the definition of complex types, which are used for elements that contain other elements or attributes.
- Simple types are used for elements that contain only text content without child elements.
- XML Schema allows you to restrict data types using facets like minLength, maxLength, minInclusive, maxInclusive, etc.

Example:

- Here's a simplified example of an XML Schema defining a bookstore structure.

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="bookstore">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="book" type="bookType" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="bookType">
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="author" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="price" type="xs:decimal"/>
  </xs:complexType>
   </xs:schema>
```

In this example, the schema defines a `bookstore` element containing multiple `book` elements, where each book has a `title`, `author`, & `price`. The `price` is defined as an attribute with a decimal data type.

11. Benefits of JSON over XML.

Ans. JavaScript Object Notation (JSON) is a lightweight text-based open standard designed for human-readable data interchange. It is derived from the JavaScript programming language for representing simple data structures & associative arrays, called objects. Despite its relationship to JavaScript, it is language-independent, with parsers available for most programming languages.

Extensible Markup Language (XML) is a set of rules for encoding documents in machine-readable form. XML's design goals emphasize simplicity, generality, & usability over the Internet.

Benefits of JSON over XML:

1. Lightweight Representation: JSON is more concise & lightweight compared to XML. It requires fewer characters to represent the same data, as it doesn't need opening & closing tags for each element.

2. Simplicity & Readability: JSON has a simpler & more straightforward structure, making it easier to read & write. It uses a key-value pair format, which is intuitive & closely resembles data structures in programming languages.

3. Data & Methods: JSON not only represents data but also allows the inclusion of methods, providing a more versatile & dynamic data format.

4. Transportation Independence: JSON is independent of the transport mechanism, making it easier to fetch data from various sources. This is particularly useful for client-side applications that may retrieve data from different domains.

5. Better Integration with JavaScript:

- JSON is a natural fit for JavaScript, making it easy to work with in web development. Objects in JSON align seamlessly with JavaScript objects, simplifying data manipulation.

6. No Need for Proxies: Unlike XML, JSON allows cross-domain requests without the need for proxy servers. This simplifies the process of fetching data from external sources in web applications.

7. Human-Readable Output: JSON is generally easier to read & understand for both humans & machines. The structure is clean & doesn't involve excessive tags.

```json
{
  "students": [
    {"name": "Alice", "age": 22},
    {"name": "Bob", "age": 25},
    {"name": "Charlie", "age": 20}
```

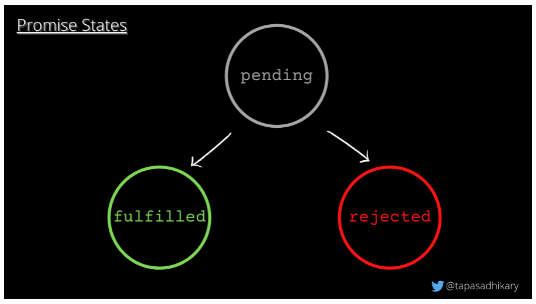
~~~
  ]
}
```xml
<!-- XML -->
<students>
  <student>
    <name>Alice</name>
    <age>22</age>
  </student>
  <student>
    <name>Bob</name>
    <age>25</age>
  </student>
  <student>
    <name>Charlie</name>
    <age>20</age>
  </student>
</students>
~~~

Chapter 2:

1. Explain promises in ES6.

Ans. A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

A Promise is in one of these states:

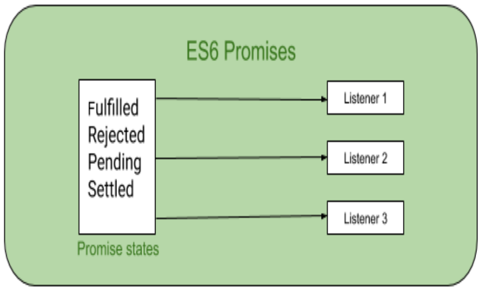


pending: initial state, neither fulfilled nor rejected.

fulfilled: meaning that the operation was completed successfully.

rejected: meaning that the operation failed.

1. ES6 & Promises:



- ES6 (ECMAScript 6) is the 6th version of ECMAScript, which standardizes JavaScript.

- Promises in ES6 facilitate asynchronous programming & act as containers for future values.

2. Understanding Promises with an Example:

- A Promise is like a receipt for an online order, representing the future delivery of a value.

- It solves issues like callback hell, which arises from nested callbacks in asynchronous JavaScript code.

3. Callback Hell: Callbacks are suitable for basic cases but can lead to code complexity, especially in web development.

4. Benefits of Promises: Promises mitigate callback hell, making code more readable & manageable.

5. Creating Promises:

- A Promise is in one of four states: fulfilled, rejected, pending, or settled.

- Syntax for creating a Promise:

```
const promise = new Promise((resolve, reject) => { /* ... */ });
```

6. Handling Promises with .then() & .catch():

- `.then()` is used for handling fulfillment & rejection.

```
promise.then((fulfilledValue) => { /* ... */ }, (error) => { /* ... */ });
```

- `.catch()` is specifically for handling errors.

```
promise.catch((error) => { /* ... */ });
```

7. Example of .creating a promise using .then() & .catch():

```
const promise = new Promise((resolve, reject) => {

  resolve('Hello, I am a Promise!');

});

promise.then((fulfilledValue) => { console.log(fulfilledValue); })

    .catch((error) => { console.log(error); });

console.log(promise);
```

2. Difference between ES5 & ES6

Ans.

Based on	ES5	ES6
Definition	ES5 is the fifth edition of the ECMAScript (a trademarked scripting language specification defined by ECMA International)	ES6 is the sixth edition of the ECMAScript (a trademarked scripting language specification defined by ECMA International)
Release	It was introduced in 2009.	It was introduced in 2015.

Data-types	ES5 supports primitive data types that are **string, number, boolean, null,** & **undefined**.	In ES6, there are some additions to JavaScript data types. It introduced a new primitive data type **'symbol'** for supporting unique values.
Defining Variables	In ES5, we could only define the variables by using the **var** keyword.	In ES6, there are two new ways to define variables that are **let** & **const**.
Loops	In ES5, there is a use of **for** loop to iterate over elements.	ES6 introduced the concept of **for...of** loop to perform an iteration over the values of the iterable objects.
Support	A wide range of communities support it.	It also has a lot of community support, but it is lesser than ES5.

Object Manipulation	ES5 is more time-consuming than ES6.	Due to destructuring & speed operators, object manipulation can be processed more smoothly in ES6.
Arrow Functions	In ES5, both **function** & **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.

```
Example: // ES5

function addES5(a, b) {

  return a + b;

}

var resultES5 = addES5(3, 7);

console.log(resultES5); // Output: 10

Example: // ES6

const addES6 = (a, b) => a + b;
const resultES6 = addES6(3, 7);

console.log(resultES6); // Output: 10
```

3. **Explain arrow function in ES6.**

Ans. Arrow functions are anonymous functions i.e. they are functions without a name and are not bound by an identifier. Arrow functions do not return any value and can be declared without the function keyword. They are also called Lambda Functions.

1. Arrow functions do not have the prototype property like this, arguments, or super.
2. Arrow functions cannot be used with the new keyword.
3. Arrow functions cannot be used as constructors.

Syntax: For Single Argument:

```
let function_name = argument1 => expression
```

For Multiple Arguments:

```
let function_name = (argument1, argument2 , ...) => expression
```

Differences of arrow functions from normal functions:

- It should not be used as a method because it lacks its own bindings to this or super.
- Within its body, it is unable to use yield.
- It doesn't be used while return statements are present.

- keyword to be targeted (that is let, var, const)
- Not suited for methods like call, apply, and bind, which all require setting a scope.
- We can't use a new keyword to create a new object.
- They are unable to be utilized as constructors.
- it doesn't have arguments object and prototype property.

Examples:

- Example 1: Multiplying two numbers using traditional & arrow function syntax.

```
// Traditional function for multiplication

function multiply(a, b) {

  return a * b;

}

console.log(multiply(3, 5));

// Output: 15

// Arrow function for multiplication

value = (a, b) => a * b;

console.log(value(3, 5));

// Output: 15
```

- Example 2: Handling multiple lines of code in an arrow function.

```


number = (a, b) => {
 c = 5;

 return (a + b) * c;

};

console.log(number(2, 3));

// Output: 25
```

- Example 3: Arrow function with no parameters.

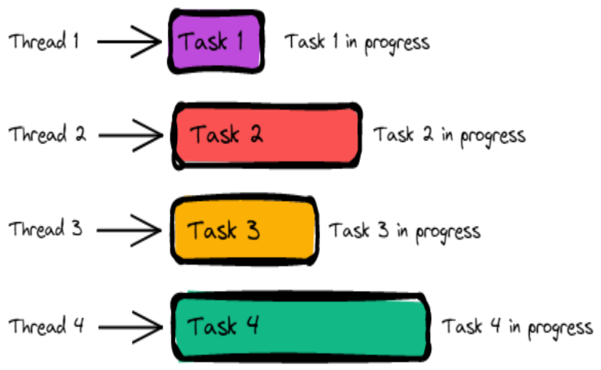
```
 const string = () => "geeksforgeeks";

 console.log(string);

// Output: geeksforgeeks
```

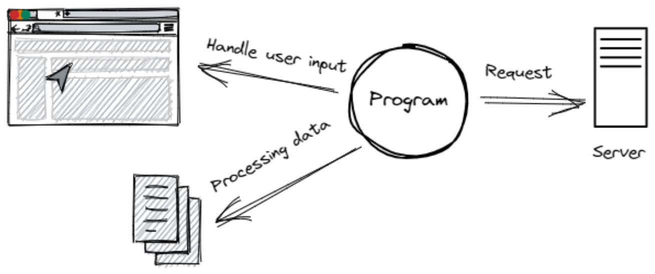
- Example 4: Using arrow functions inside another function (e.g., `map`).

```
// Initializing an array of strings

   let array = ["sam", "sarah", "john"];

   // Map function used to find the length of strings

   let lengths = array.map((string) => string.length);

   console.log(lengths);

// Output: [3, 5, 4]
```

4. Explain asynchronous programming in detail.

Ans. Asynchronous programming is a programming paradigm that allows a program to perform tasks concurrently, handling multiple operations without waiting for each one to complete before moving on to the next. It is particularly useful for





tasks that may be time-consuming. Asynchronous programming enables a program to remain responsive and efficient by initiating tasks, continuing to execute other code, and handling the results of the tasks once they are completed.

Key Concepts:

1. Concurrency:

   - Concurrency is the ability of a program to execute multiple tasks at the same time.

   - Asynchronous programming facilitates concurrency by allowing tasks to overlap in execution, enhancing the overall performance of the program.

2. Callbacks:

   - Callbacks are functions passed as arguments to other functions. They are executed once a specific task is completed.

   - In asynchronous programming, callbacks are commonly used to handle the results of asynchronous operations.

3. Promises:

- Promises are objects representing the eventual completion or failure of an asynchronous operation.

   - They provide a cleaner and more structured way to work with asynchronous code compared to callbacks.

4. Event Loop:

   - In the context of JavaScript and web browsers, the event loop is a fundamental concept in asynchronous programming.

   - It continuously checks the message queue for tasks to execute, allowing non-blocking execution.

Example of Asynchronous Programming using setTimeout:

```
console.log("Start of script");

setTimeout(function() {

  console.log("First timeout completed");

}, 2000);

console.log("End of script");



Output:

Start of script

End of script

First timeout completed
```

In this example:

- The `setTimeout` function initiates a function to be executed after a specified time (2 seconds).

- The function inside `setTimeout` runs asynchronously, allowing the program to continue executing the next line of code without waiting for the timeout to complete.

- The program prints "First timeout completed" after 2 seconds while proceeding with the rest of the code.

Benefits of Asynchronous Programming:

1. Improved Performance:

   - Asynchronous programming prevents blocking, enabling a program to perform multiple tasks simultaneously.

   - This is crucial for tasks like fetching data from external sources or handling user interactions without freezing the application.

2. Responsiveness:

   - Asynchronous code ensures that the application remains responsive to user input even when performing time-consuming operations.

   - This contributes to a better user experience.

3. Efficient Resource Utilization:

- By allowing tasks to run concurrently, asynchronous programming optimizes the utilization of system resources.

4. Scalability:

- Asynchronous patterns are essential for building scalable systems, especially in scenarios where many clients or users interact with a system concurrently.

5. What are events in JS? Explain different types of events in JSX.

Ans. Events are things that happen in the system you are programming — the system produces (or "fires") a signal of some kind when an event occurs, and provides a mechanism by which an action can be automatically taken (that is, some code running) when the event occurs. Events are fired inside the browser window, and tend to be attached to a specific item that resides in it. This might be a single element, a set of elements, the HTML document loaded in the current tab, or the entire browser window. There are many different types of events that can occur.

For example:

- The user selects, clicks, or hovers the cursor over a certain element.
- The user chooses a key on the keyboard.
- The user resizes or closes the browser window.
- A web page finishes loading.
- A form is submitted.
- A video is played, paused, or ends.
- An error occurs.

Event Types & Examples: Different types of events can occur in a browser window, covering a spectrum of user interactions and system actions. Examples include user interactions like selection, clicks, or cursor hovering, keyboard inputs, browser-related actions such as window resizing or closing, page lifecycle events like page finishing loading, form submissions, media events (e.g., video playing, pausing, or ending), and error events.

Event Handling Mechanism: Events are attached to specific items within the browser window, such as a single element, a set of elements, the HTML document, or the entire browser window. To respond to an event, an event handler is attached—a block of code, typically a JavaScript function, that runs when the event occurs. This process of defining and associating this code with an event is known as registering an event handler.

```
// Syntax for registering an event handler

element.addEventListener('eventType', eventHandlerFunction);
```

Event Handlers & Listeners: Event handlers, sometimes referred to as event listeners, are interchangeable terms. The listener "listens" for the occurrence of an event, while the handler is the code executed in response to the event. JavaScript provides a robust mechanism to handle events and execute custom code based on user actions or system changes.

```
// Syntax for adding an event listener

element.addEventListener('eventType', eventHandlerFunction);
```

Dynamic Interface in JavaScript: JavaScript events play a crucial role in creating dynamic interfaces in web development. They enable the creation of responsive and interactive web pages by allowing developers to execute code in response to user interactions, providing a seamless user experience.

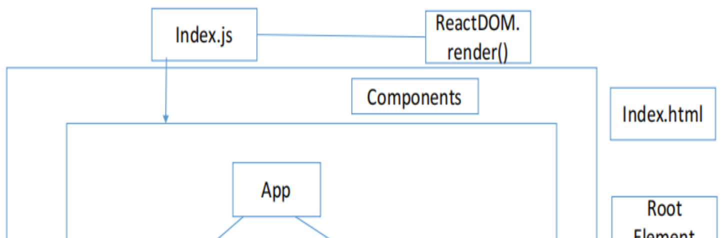
DOM & Event Propagation: Events in JavaScript are hooked to elements in the Document Object Model (DOM). By default, events use bubbling propagation, moving upwards in the DOM from children to parent elements. Understanding event propagation is vital for effective event handling in complex web applications.

Binding Events: Events can be bound either inline directly within the HTML or in an external script. Inline binding involves adding event attributes directly to HTML elements. On the other hand, external script binding is done by writing JavaScript code separately to handle events, offering a more organized and maintainable approach to event handling in larger projects.

```
<!-- Syntax for inline event binding -->

<button onclick="myFunction()">Click me</button>

<!-- Syntax for external script event binding -->

<script>

  element.addEventListener('eventType', eventHandlerFunction);

</script>
```

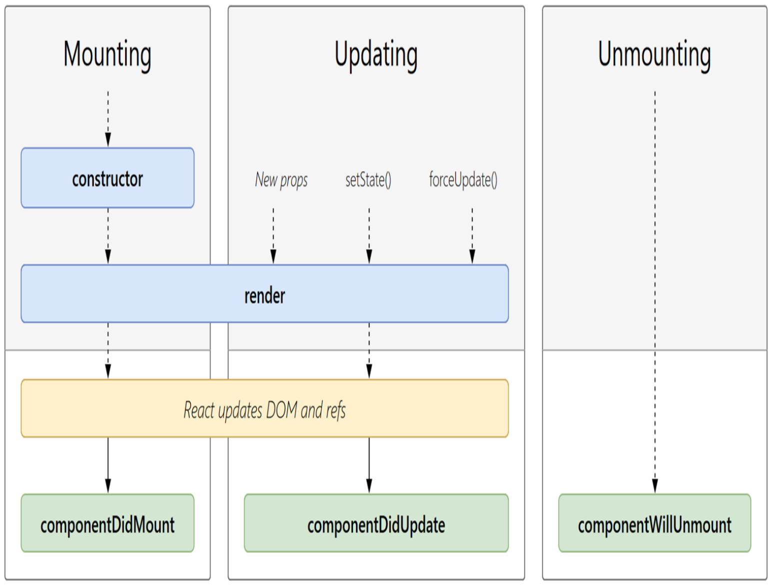
Chapter 3 & 4:

1. Explain react component lifecycle.



Ans. A React component is a reusable, self-contained unit of code that represents a part of the user interface. Components are the building blocks of a React application, & they encapsulate the logic & view of a specific part of the UI. Each component can manage its own state & properties, & it can be composed with other components to create complex user interfaces.

React Component Lifecycle: The React component lifecycle refers to the various phases a React component goes through from its creation to its removal from the DOM. Each phase has lifecycle methods associated with it, allowing developers to execute code at specific points in the component's lifecycle.



Here are the main phases of the React component lifecycle:

1. Initialization:

 - constructor(): This method is called when a component is being constructed. It initializes the component's state & binds event handlers.

```jsx
class MyComponent extends React.Component {
    constructor(props) {
      super(props);
      this.state = {
        // initialize state
      };
    }
  }
```

2. Mounting:

   - componentWillMount(): Deprecated.

   - render(): This method is mandatory & responsible for rendering the component. It returns the React elements.

```jsx
  render() {
    return <div>Hello, World!</div>;
  }
```

   - componentDidMount(): This method is called after the component has been rendered to the DOM. It is often used for data fetching or setting up subscriptions.

```jsx
componentDidMount() {
  // perform actions after component is mounted
}
```

3. Updating:

   - componentWillUpdate(): Deprecated.

   - render(): Re-render the component.

   - componentDidUpdate(): Called after the component's updates are flushed to the DOM.

```jsx
componentDidUpdate(prevProps, prevState) {
  // perform actions after component updates
}
```

4. Unmounting:

   - componentWillUnmount(): Called before the component is removed from the DOM. It is used to perform cleanup.

```jsx
componentWillUnmount() {
     // perform cleanup before component is unmounted
   }
```

Example:

```jsx
import React from 'react';
class LifecycleExample extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      message: 'Hello, React Lifecycle!',
    };
  }
  componentDidMount() {
    // This is a good place for data fetching
    console.log('Component has mounted');
  }
  componentDidUpdate() {
    // This is invoked after a state or prop update
    console.log('Component has updated');

  }

  componentWillUnmount() {

    // Cleanup before component is unmounted

    console.log('Component will unmount');

  }

  render() {

    return <div>{this.state.message}</div>;

  }

}

export default LifecycleExample;
```

2. **Explain redux architecture in detail.**

Ans. Redux is an open-source JavaScript library used to manage application state. React uses Redux for building the user interface.

React Redux is the official React binding for Redux. It allows React components to read data from a Redux Store, & dispatch Actions to the Store to update data. Redux helps apps to scale by providing a sensible way to manage state through a unidirectional data flow model. React Redux is conceptually simple. It subscribes to the Redux store, checks to see if the data which your component wants have changed, & re-renders your component. Redux was inspired by Flux. Redux studied the Flux architecture & omitted unnecessary complexity.

1. Redux does not have Dispatcher concept.

2. Redux has an only Store whereas Flux has many Stores.

3. The Action objects will be received & handled directly by the Store.

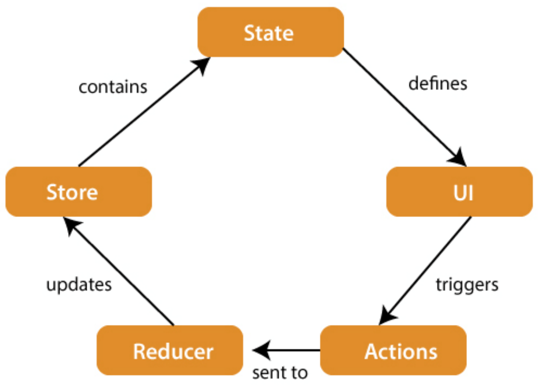
The main reason to use React Redux are:

1. React Redux is the official UI bindings for react Application. It is kept up-to-date with any API changes to ensure that your React components behave as expected.

2. It encourages good 'React' architecture.

3. It implements many performance optimizations internally, which allows components to re-render only when it actually needs.

The components of Redux architecture are explained below.



STORE: A Store is a place where the entire state of your application lists. It manages the status of the application & has a dispatch(action) function. It is like a brain responsible for all moving parts in Redux.

ACTION: Action is sent or dispatched from the view which are payloads that can be read by Reducers. It is a pure object created to store the information of the user's event. It includes information such as type of action, time of occurrence, location of occurrence, its coordinates, & which state it aims to change.

REDUCER: Reducer read the payloads from the actions & then updates the store via the state accordingly. It is a pure function to return a new state from the initial state.

Redux Installation:

```
 npm install redux react-redux --save
```

3. What are refs? When to use refs & when not to use refs?

Ans. Refs are a function provided by React to access the DOM element & the React element that you might have created on your own. They are used in cases where we want to change the value of a child component, without making use of props & all. They have wide functionality as we can use callbacks with them.

Creating refs in React

Refs can be created using React.createRef() function & attached it with the React element via the ref attribute. When a component is constructed the Refs are commonly assigned to an instance property so that they can be referenced in the component.

```
class MyComponent extends React.Component {
```

```
constructor(props) {

super(props);

this.myCallRef = React.createRef();

}

render() {

return <div ref={this.myCallRef} />;

}

}
```

Accessing Refs in React

In React, A reference to the node becomes accessible at the current attribute of the ref when a ref is passed to an element in the render.

const node = this.myCallRef.current;

When to use refs

- Helpful when using third-party libraries.
- Helpful in animations.
- Helpful in managing focus, media playback, & text selection.

When not to use refs

- Should not be used with functional components because they don't have instances.

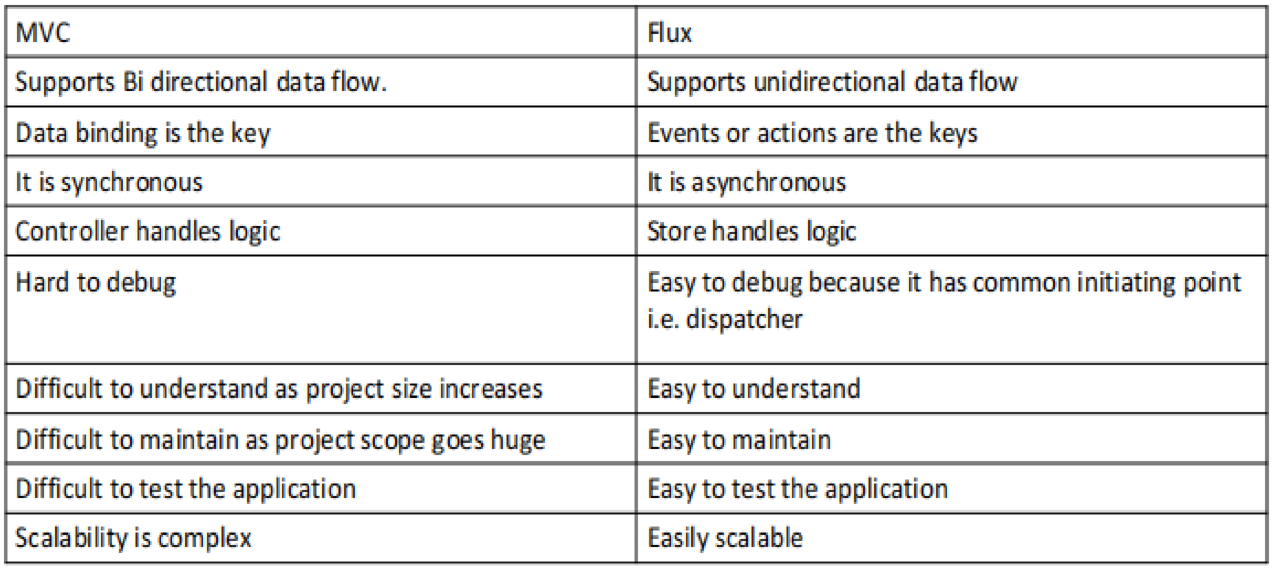
- Not to be used on things that can be done declaratively.
- When using a library or framework that provides its own methods for managing suchas Redux or MobX.

4. Compare MVC & Flux.

Ans. Example:

MVC: A user clicks a "Increment" button triggering an event handled by the controller. The controller updates the count in the model, & the view reflects the updated count.

Flux: A user clicks a "Increment" button triggering an action. The dispatcher dispatches this action to the store. The store updates the count & notifies the React component (view) subscribed to it, causing a re-render with the updated



MVC	Flux
Supports Bi directional data flow.	Supports unidirectional data flow
Data binding is the key	Events or actions are the keys
It is synchronous	It is asynchronous
Controller handles logic	Store handles logic
Hard to debug	Easy to debug because it has common initiating point i.e. dispatcher
Difficult to understand as project size increases	Easy to understand
Difficult to maintain as project scope goes huge	Easy to maintain
Difficult to test the application	Easy to test the application
Scalability is complex	Easily scalable

count.

Chapter 5:

1. **Explain REPL.**

Ans. The term REPL stands for Read Eval Print & Loop. It specifies a computer environment like a window console or a Unix/Linux shell where you can enter the commands & the system responds with an output in an interactive mode.

REPL Environment

Node.js or node come bundled with the REPL environment. Each part of the REPL environment has a specific work.

Read: It reads user's input; parses the input into JavaScript data-structure & stores it in memory.

Eval: It takes & evaluates the data structure.

Print: It prints the result.

Loop: It loops the above commands until user press ctrl-c twice.

Node.js REPL Commands

ctrl + c – terminate the current command.

ctrl + c twice – terminate the Node REPL.

ctrl + d – terminate the Node REPL.

Up/Down Keys – see command history & modify previous commands.

tab Keys – list of current commands.

.help – list of all commands.

.break – exit from multiline expression.

.clear – exit from multiline expression.

.save filename – save the current Node REPL session to a file.

.load filename – load file content in current Node REPL session.

Example :-

```
>x = 10

10

>var y = 10

Undefined

>x + y

20

>Console.log("Hello World")

Hello World

Undefined
```

2. **What is nodejs? Explain features of NodeJs. State different types of NodeJs Modules.**

Ans. Node.js is a JavaScript runtime built on the V8 JavaScript engine. It allows developers to execute JavaScript code server-side, providing a runtime environment for building scalable & high-performance network applications. Node.js has gained popularity for its efficiency in handling asynchronous operations.

Features of Node.js:

1. Non-blocking I/O: Node.js is designed to handle a large number of simultaneous connections without blocking, making it suitable for real-time applications.

2. Event-Driven: It follows an event-driven architecture, utilizing events to handle asynchronous operations efficiently.

3. Fast Execution: Powered by the V8 engine, Node.js executes JavaScript code swiftly, making it suitable for building fast & scalable applications.

4. Single Programming Language: Allows developers to use JavaScript for both server-side & client-side development, promoting a consistent language across the stack.

5. NPM (Node Package Manager): The built-in package manager NPM simplifies the installation & management of third-party packages & libraries.

6. Scalability: Supports horizontal scaling by allowing the addition of more machines to the existing system, improving performance.

7. Cross-Platform: Node.js is compatible with various operating systems, including Windows, macOS, & Linux.

Node.js Modules:

Node.js follows a modular architecture, & modules are a fundamental concept. Modules encapsulate code within a file & expose specific functionalities. There are different types of modules:

1. Core Modules: These are built-in modules provided by Node.js, such as `fs` (file system), `http` (HTTP server), & `util` (utility functions).

   Example:

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

fs.readFile('file.txt', 'utf8', (err, data) => {

  if (err) throw err;

  console.log(data);

});
```

2. Third-Party Modules: Developed by the Node.js community & available through NPM. Developers can install & use these modules in their projects.

   Example:

```
npm install lodash
```

~~~
```javascript

const _ = require('lodash');

console.log(_.sum([1, 2, 3, 4]));
~~~

3. Local Modules: Modules created by developers for specific projects. These modules encapsulate functionalities & can be used within the project.

Example:

~~~
```javascript

// math.js

const add = (a, b) => a + b;

module.exports = { add };

```javascript

// main.js

const math = require('./math');

console.log(math.add(2, 3));
~~~

Node.js modules promote code organization, reusability, & maintainability in applications.

3. Explain event loop in nodejs.

Ans.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 & wherever possible.

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

Features of Event Loop:

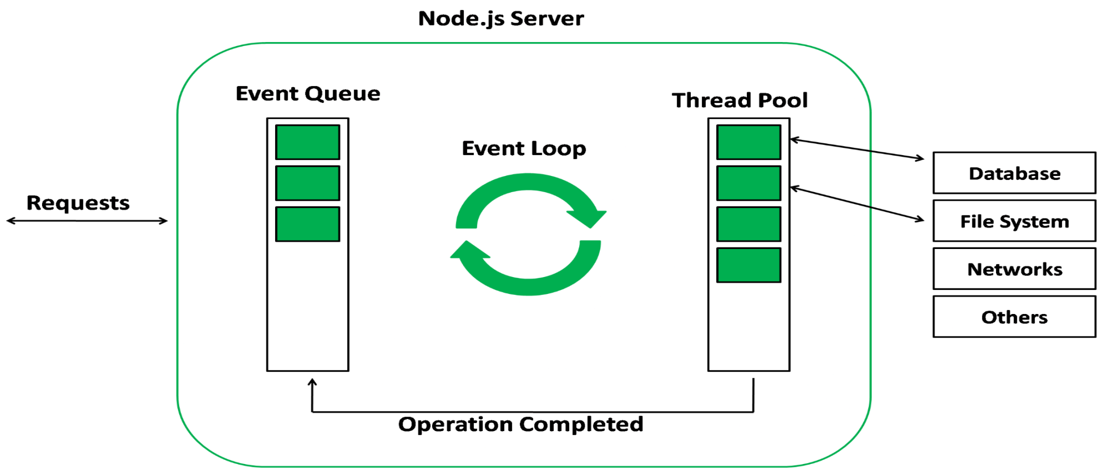
An event loop is an endless loop, which waits for tasks, executes them, & then sleeps until it receives more tasks.

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 & promises.

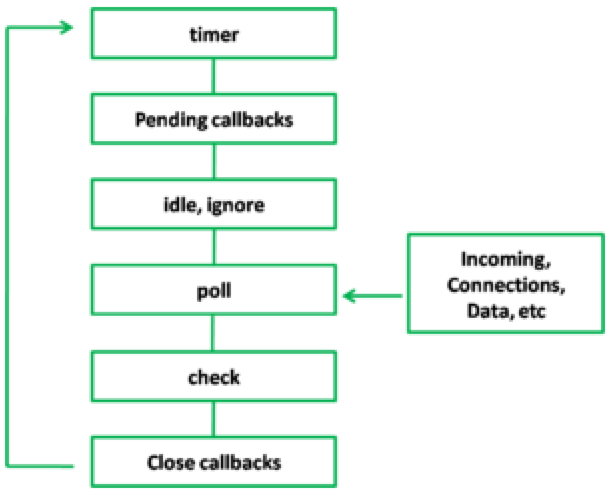
The event loop executes the tasks starting from the oldest first.

Working of 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:



Example:
```javascript

const fs = require('fs');

// Example 1: Reading a file asynchronously

fs.readFile('example.txt', 'utf8', (err, data) => {

  if (err) throw err;

  console.log('File Content:', data);

  // Example 2: setTimeout as a non-blocking operation

  setTimeout(() => {

    console.log('Timeout Completed');

    // Example 3: setInterval as a recurring non-blocking operation
    const intervalId = setInterval(() => {
      console.log('Interval Operation');
    }, 1000);
    // Example 4: Simulating an event (e.g., HTTP request)
    setTimeout(() => {
      clearInterval(intervalId);
      console.log('Event Completed');
    }, 5000);
  }, 2000);
});
console.log('Event Loop Demonstration');
```

This example covers the following:

1. Reading a file asynchronously using `fs.readFile`.

2. Using `setTimeout` to demonstrate a non-blocking operation.

3. Utilizing `setInterval` to create a recurring non-blocking operation.

4. Simulating an event (e.g., an HTTP request) with another `setTimeout`.

4. What are buffers & streams in Nodejs. Explain with example.

Ans. Buffer Concept:

1. A buffer is a temporary space in memory (typically RAM) that stores binary data to prevent data loss during transfers, following a FIFO system.

2. In Node.js, the Buffer module in its core allows manipulation of these memory spaces, especially when working with binary data at the network level.

3. Buffers play a crucial role in modules like File System or HTTP, handling temporary data flow implicitly based on events & Buffer/Stream manipulation.

Important Features of Buffers:

1. Buffers are stored in a sequence of integers.

2. Once created, Buffers cannot be resized.

Example: Creating a Buffer in Node.js:

```javascript
const { Buffer } = require('buffer');

const buf6 = Buffer.from('hello');

console.log(buf6);
```

Output:

```
<Buffer 68 65 6c 6c 6f>

hello
```

```
// [0x68, 0x65, 0x6c, 0x6c, 0x6f] (hexadecimal notation)
```

Reading Stored Content:

```javascript
const text = buf6.toString();

console.log(text);  // hello
```

Stream Concept:

1. Streams are information flows used for transmitting binary data efficiently, handling network communications or any data exchange by breaking information into smaller pieces.

2. Streams work sequentially & transmit data in "chunks," relying on buffers to manage content.

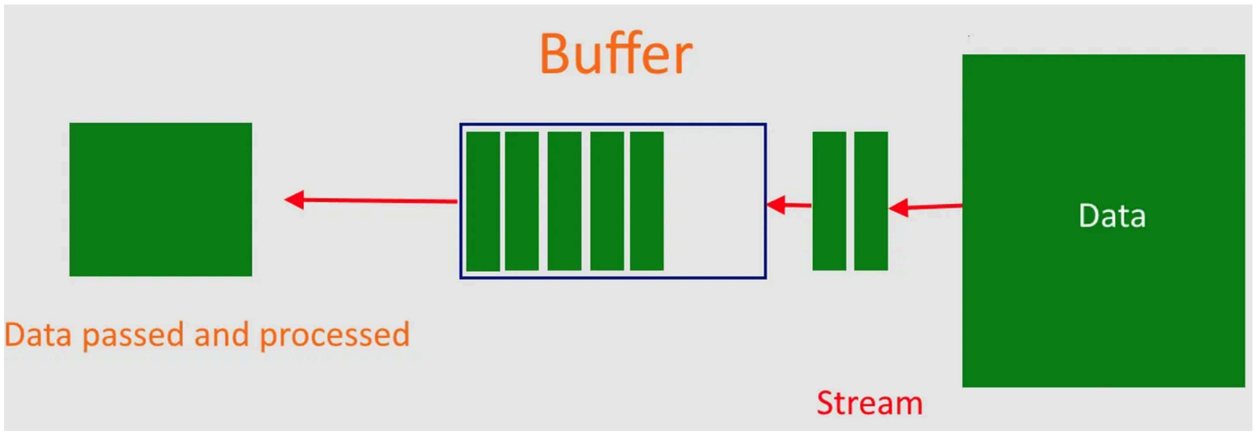
3. Streams are effective for processing large amounts of information, allowing the handling of data in smaller, manageable pieces.

Important Features of Streams:

1. Streams work sequentially.

2. Information is transmitted through "chunks" (pieces).

3. Streams rely on buffers to manage content.



Node.js Stream Module:

1. Node.js has a built-in `stream` module to manipulate flows, offering different types of streams: Writable, Readable, Duplex, & Transform.

2. Examples of Node.js applications using streams include HTTP servers reading streams in requests (Readable) & writing streams in responses (Writable).

Example: Creating a Readable Stream in Node.js:

```javascript

const Stream = require('stream');

const readableStream = new Stream.Readable();

readableStream.push('Hello');

readableStream.push(' ');

readableStream.push('World');

readableStream.push(null);
```

```
async function getContentReadStream(readable) {
  for await (const chunk of readable) {
    console.log(chunk);
    console.log(chunk.toString());
  }
}
getContentReadStream(readableStream);
```

Output:

```
<Buffer 48 65 6c 6c 6f 20 57 6f 72 6c 64>
Hello World
```

Chapter 6:

1. **Explain routing in express.js along with an example.**

Ans. Routing in ExpressJS involves defining how an application responds to client requests for specific URIs (endpoints). Each route is associated with an HTTP method, & Express provides methods like `app.get()`, `app.post()`, etc., to handle different types of requests.

### Example:

```javascript
var express = require('express');
var app = express();
// Define a route for the homepage
app.get('/', function(req, res) {
  console.log("Homepage Request");
  res.send("Hello World");
});
// Define a route for /apple
app.get('/apple', function(req, res) {
  console.log("Request from /apple");
  res.send("Apple");
});
// Start the server on port 3000
app.listen(3000);
console.log ('server is running on port 3000');
```

### Explanation:

1. Creating an Express Application:

- `var express = require('express');` : Import the Express module.

- `var app = express();` : Create an instance of the Express application.

2. Defining Routes:

- `app.get('/', function(req, res) { ... });` : Handle GET requests for the root path ('/') or homepage. When a user accesses the root path, the function inside is executed.

- `app.get('/apple', function(req, res) { ... });` : H&le GET requests for the '/apple' path. When a user accesses '/apple', the corresponding function is executed.

3. Handling Requests:

- `function(req, res) { ... }` : This is the callback function that gets executed when the specified route is accessed. It takes two parameters: `req` (request) & `res` (response).

- `console.log("Homepage Request");` : Log a message to the console when the homepage route is accessed.

- `res.send("Hello World");` : Send the "Hello World" response to the client when the homepage route is accessed.

- Similar logic applies to the '/apple' route.

4. Starting the Server:

- `app.listen(3000);` : Start the server on port 3000.

2. Short note on express.js

Ans. Express.js is a web application framework for Node.js, simplifying the development of web & mobile applications. It serves as a layer on top of Node.js, providing features for server management & routing.

- Efficiency: Reduces coding time significantly, making web & mobile app development efficient.

- JavaScript: Enables easy entry for new developers into web development, leveraging the familiarity of JavaScript.

- Time-Efficient: Built to save time, with features like middleware, routing, & templating engines.

Key Features:

1. Fast Server-Side Development: Leverages Node.js features for rapid server-side development.

2. Middleware: Utilizes request handlers for the application's request-response cycle.

3. Routing: Efficiently handles client requests based on endpoint URLs.

4. Templating: Offers templating engines for dynamic content creation on web pages.

5. Debugging: Simplifies debugging by pinpointing the exact location of errors.

Advantages:

- Customization: Unopinionated framework, allowing customization.

- Middleware Usage: Facilitates efficient request h&ling with middleware.

- Single Language: Enables the use of a single language (JavaScript) for both frontend & backend development.

- Database Integration: Fast integration with databases like MySQL, MongoDB.

Limitations:

- Structural Organization: Lack of a structured way to organize code may lead to non-understandable code.

- Callback Issues: Challenges with callback functions.

- Error Messages: Error messages can be difficult to understand.

Example:
```javascript
const express = require('express');

const app = express();

app.get('/', (req, res) => {

  res.send('Hello, Express!');

});

app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

Companies Using Express.js:

- Netflix

- IBM

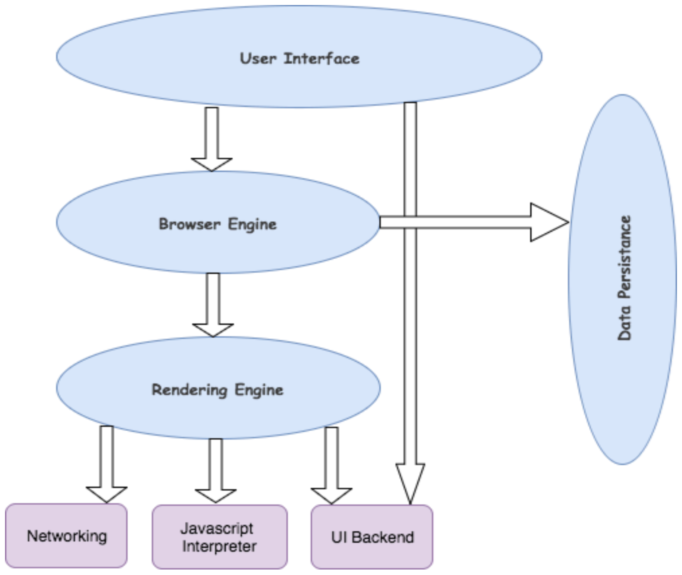
- eBay

- Uber

BrainHeaters:

1. **Explain the working of web browsers with the help of diagram.**

Ans. Web browsers play a pivotal role in accessing & rendering content from the World Wide Web. They facilitate the retrieval & display of web pages, documents, images, & more. Let's delve into their functions & components.



1. Introduction to Browser Functionality:

- Web browsers are sophisticated software applications designed to interact with the World Wide Web, retrieving & presenting various types of content, including text, images, videos, & other files.

- Operating on a client/server model, the browser acts as a client, reaching out to web servers to request information. Subsequently, the web server responds by sending the requested data back to the browser, which then displays it on the user's device.

2. Browsing Capabilities:

- Modern browsers are comprehensive software suites that go beyond simply rendering HTML. They can interpret & display a wide range of web technologies, including JavaScript, AJAX, & various applications.

- Plug-ins further enhance browser capabilities, allowing the integration of multimedia content such as sound & video. This versatility makes browsers essential tools for diverse tasks beyond traditional web browsing.

3. User Interface:

- The user interface serves as the interactive space where users engage with the browser. It encompasses a range of elements, including the address bar, navigation buttons, bookmarks, & additional features.

- It provides users with the means to input comm&s, navigate through web pages, & access various functionalities offered by the browser.

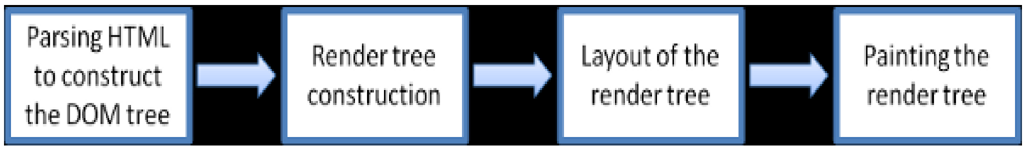
4. Browser Engine:

- Sitting between the user interface & the rendering engine, the browser engine acts as a coordinator. It interprets user inputs from the interface & communicates instructions to the rendering engine accordingly.

- This component plays a crucial role in managing the flow of information between the browser's user-facing elements & the backend processes responsible for rendering web content.

5. Rendering Engine:

- The rendering engine is at the core of displaying web content. It interprets HTML, XML, & CSS files, converting them into a visual layout that users can interact with.


Parsing HTML to construct the DOM tree
Render tree construction
Layout of the render tree
Painting the render tree

- Different browsers employ different rendering engines (e.g., Trident, Gecko, Blink, WebKit), each with its own set of rules for rendering web content.

6. Networking:

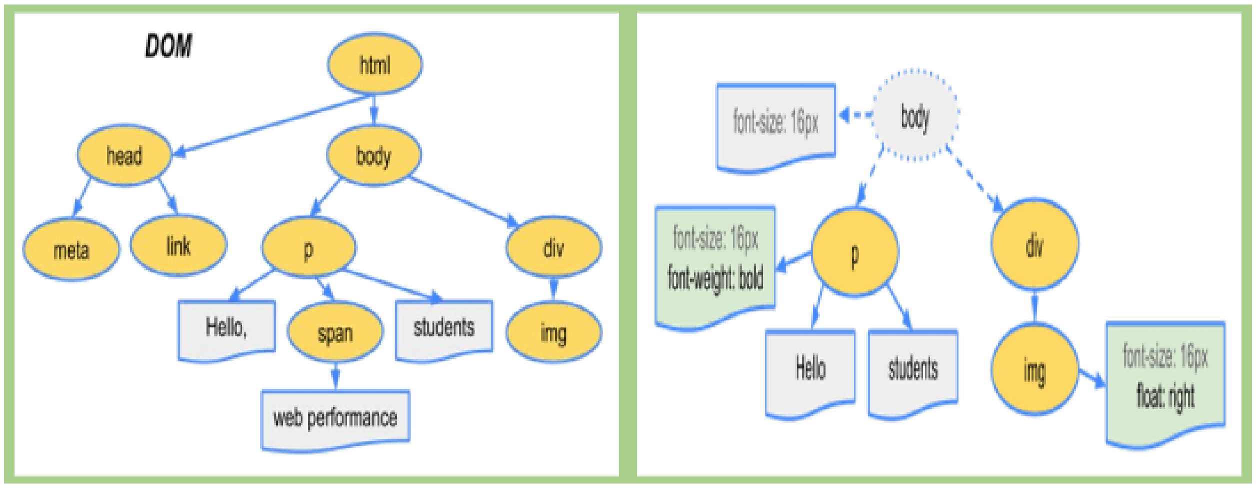
- The networking component h&les the retrieval of URLs using st&ard internet protocols like HTTP or FTP. It facilitates communication between the browser & web servers.

- In addition to retrieving content, the networking component plays a role in implementing security measures & may include features like caching to optimize performance by storing previously retrieved documents.

7. Rendering Engine Process:

- Content Retrieval: The networking layer retrieves the contents of requested documents from web servers, delivering them to the rendering engine in manageable chunks.

- DOM Tree Construction: The rendering engine parses HTML chunks, creating a Document Object Model (DOM) tree. This tree represents the document's structure, allowing the browser to underst& & manipulate the content.



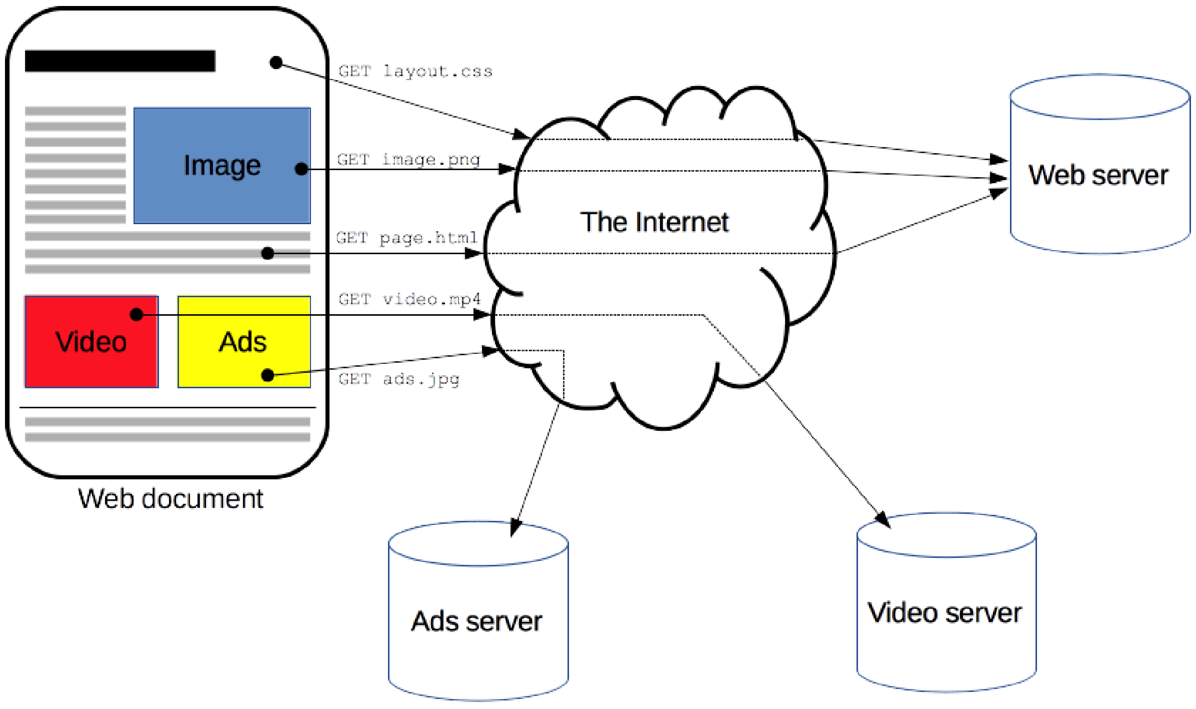
- Render Tree Construction: Simultaneously, a render tree is constructed, which represents the visual elements of the document in the order they will appear on the screen.

- Layout Process: The render tree undergoes a layout process, determining the exact position & size of each visual element. This step ensures the correct presentation of content.

- Painting Stage: The rendering engine traverses the render tree, invoking the "paint()" method for each renderer. This process results in the final visual representation of the document on the user's screen, utilizing a UI backend layer for the painting process.

2. **Describe HTTP protocol in detail.**

Ans. HTTP is a protocol for fetching resources such as HTML documents. It is the foundation of any data exchange on the Web & it is a client-server protocol, which means requests are initiated by the recipient, usually the Web browser. A complete document is reconstructed from the different sub-documents fetched, for instance, text, layout description, images, videos, scripts, & more.



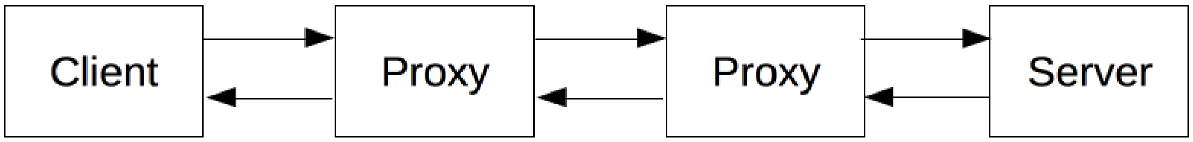
- Foundation of Web Data Exchange:

  - HTTP is a protocol facilitating the fetching of resources, such as HTML documents, forming the backbone of data exchange on the Web.

- It operates on a client-server model where requests are initiated by the recipient, typically the Web browser.

- Complete documents are reconstructed from various sub-documents, including text, layout descriptions, images, videos, scripts, etc.

Components of HTTP:



- Client-Server Protocol:

  - HTTP is a client-server protocol where the user-agent (typically the Web browser) sends requests, & servers respond with answers.

  - Proxies, acting as gateways or caches, may exist between the client & server, h&ling various operations.

- User-Agent (Client):

  - The user-agent, often the Web browser, initiates requests to fetch resources like HTML documents, scripts, layout information (CSS), & sub-resources (images, videos).

  - The browser compiles these resources to present a complete document or Web page to the user.

- Web Server (Server):

- The server serves requested documents to the client, appearing as a single machine but may comprise multiple servers, sharing the load or h&ling specific functions.

- Proxies:

  - Proxies relay HTTP messages between the Web browser & server.

  - Operating at different layers, proxies can be transparent, forwarding requests without alteration, or non-transparent, modifying requests before reaching the server.

  - Proxies serve functions like caching, filtering, load balancing, authentication, & logging.

Characteristics/Aspects of HTTP:

- HTTP is Simple & Human-Readable:

  - Messages are designed to be simple & human-readable, aiding testing & reducing complexity.

  - Readability allows easy underst&ing by humans.

- HTTP is Extensible:

  - HTTP headers facilitate extensibility, enabling easy experimentation & introduction of new functionalities.

- HTTP is Stateless but Not Sessionless:

  - HTTP is stateless, with no link between successive requests on the same connection.

  - However, sessions can be created using mechanisms like HTTP cookies.

Common Features Controllable with HTTP:

- Caching:

  - HTTP controls how documents are cached, with servers instructing proxies & clients about what to cache & for how long.

- Authentication:

  - HTTP supports authentication, protecting specific pages to allow only authorized user access.

- HTTP Cookies:

  - HTTP cookies link requests with server state, enabling sessions in a stateless protocol.

HTTP Messages:

- HTTP messages (HTTP/1.1 & earlier) are human-readable, while HTTP/2 embeds messages into binary structures for optimizations like header compression & multiplexing.

- Two types of messages exist: requests & responses, each with its own format.

HTTPS (Hypertext Transfer Protocol Secure):

- Secure Version of HTTP:

  - HTTPS uses the TLS protocol for encryption & authentication, ensuring secure transmission of sensitive data.

  - Vital for securing online activities like shopping, banking, & remote work.

Differences Between HTTP & HTTPS:

- Encryption:

  - HTTPS adds encryption (TLS) to prevent eavesdropping & man-in-the-middle attacks, securing data transmission.

- Authentication:

  - HTTPS incorporates robust authentication via the TLS protocol.

- Integrity:

  - HTTPS ensures integrity by including a digital signature with each document, proving it hasn't been altered in transit.

- Safety & Browsing:

  - HTTPS is safer for browsing & online activities, providing a secure protocol compared to the vulnerability of HTTP.

3. Explain React router & single page applications.

Ans. **React Router:**

React Router is a standard library for routing in React applications. It enables the navigation among views of various components in a React Application, allows changing the browser URL, & keeps UI in sync with the URL.

Key Concepts:

1. BrowserRouter:

   - The `<BrowserRouter>` component is used to set up the router. It utilizes the HTML5 history API to keep the UI in sync with the URL.

```
  import { BrowserRouter as Router } from 'react-router-dom';

  ReactDOM.render(

    <Router>

      {/* Your application components */}

    </Router>,

    document.getElementById('root')

  );
```

2. Route:

   - `<Route>` is a declarative way of rendering components based on the URL.

```
  import { Route } from 'react-router-dom';

  const Home = () => <div>Home Page</div>;
  ReactDOM.render(
    <Router>
      <Route path="/" component={Home} />
    </Router>,
    document.getElementById('root')
  );
```

3. Link:

  - `<Link>` provides a way to navigate to different views.

```
  import { Link } from 'react-router-dom';
  const Navigation = () => (
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
    </nav>
  );
```

4. Switch:

  - `<Switch>` renders the first `<Route>` or `<Redirect>` that matches the current location.

```
  import { Switch, Route } from 'react-router-dom';

  ReactDOM.render(
    <Router>
      <Switch>
        <Route path="/" component={Home} />
        <Route path="/about" component={About} />
      </Switch>
    </Router>,
    document.getElementById('root')
  );
```

Example Code:

Here's a concise example demonstrating React Router in action:

```
// App.js

import { BrowserRouter as Router, Route, Link, Switch } from 'react-router-dom';

const Home = () => <div>Home Page</div>;

const About = () => <div>About Page</div>;

const NotFound = () => <div>404 Not Found</div>;

const Navigation = () => (
  <nav>
    <Link to="/">Home</Link>
    <Link to="/about">About</Link>
  </nav>
);
const App = () => (
  <Router>
    <Navigation />
    <Switch>
      <Route path="/" exact component={Home} />
      <Route path="/about" component={About} />
      <Route component={NotFound} />
    </Switch>
  </Router>
);
export default App;
```

In this example, `Home`, `About`, & `NotFound` are components that get rendered based on the URL. The `<Navigation>` component provides links to navigate between different views.

Single Page Applications (SPAs):

A Single Page Application (SPA) is a web application or website that interacts with the user by dynamically rewriting the current page rather than loading entire new pages from the server. SPAs use AJAX & HTML5 to create fluid & responsive user experiences.

Key Characteristics:

1. Dynamic Loading:

   - SPAs load only the necessary parts of the page & update the content dynamically.

2. Smooth Transitions:

   - Navigation in SPAs often involves smooth transitions & animations, providing a seamless user experience.

3. No Full Page Reloads:

   - Traditional web applications often reload the entire page when navigating to a new view. SPAs, on the other h&, load only the required content, avoiding full page reloads.

4. Rich Interactivity:

   - SPAs provide a high level of interactivity, resembling desktop applications. User actions trigger updates to the UI without requiring a trip to the server.

5. Better Performance:

   - SPAs can provide a faster user experience by reducing the amount of data transferred between the client & server.

Example Code:

A simple React SPA example using React Router:

```
// App.js

import { BrowserRouter as Router, Route, Link } from 'react-router-dom';

const Home = () => <div>Home Page</div>;

const About = () => <div>About Page</div>;

const Navigation = () => (

  <nav>

    <Link to="/">Home</Link>

    <Link to="/about">About</Link>

  </nav>

);

const App = () => (
  <Router>
    <Navigation />
    <Route path="/" exact component={Home} />
    <Route path="/about" component={About} />
  </Router>
);
export default App;
```

In this SPA example, the components `Home` & `About` are rendered dynamically based on the URL without triggering a full page reload. The navigation is h&led smoothly using

4. Explain Bundling the application

Ans. Bundling in the context of web development refers to the process of combining multiple files, typically JavaScript or CSS files, into a single file. The purpose of bundling is to optimize the performance of a web application by reducing the number of HTTP requests required to load the page. It also helps in managing dependencies, improving code organization, & ensuring that the application loads more efficiently.

Key Concepts:

1. File Concatenation:

- Bundling involves concatenating multiple files into a single file. For example, if a web application has several JavaScript files, a bundler combines them into one file.

2. Dependency Resolution:

   - Bundlers analyze the dependencies between files & ensure that they are included in the correct order. This is crucial for maintaining the functionality of the application.

3. Minification:

   - In addition to bundling, many tools also perform minification. Minification involves removing unnecessary characters, whitespace, & renaming variables to reduce the size of the bundled file.

4. Code Splitting:

   - Code splitting is a technique where different parts of the application are bundled separately. This can be beneficial for larger applications, allowing the loading of only the required code for a specific page or feature.

Example Code:

Let's consider a simple example using a JavaScript bundler called Webpack.

Step 1: Install Webpack:

```
npm install webpack webpack-cli --save-dev
```

Step 2: Create Files:

Assume you have two JavaScript files, `file1.js` & `file2.js`.

```
// file1.js

const greet = () => console.log('Hello');

// file2.js

const name = 'World';

greet();

console.log(`Greetings, ${name}!`);
```

Step 3: Create a Configuration File (webpack.config.js):

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

module.exports = {

  entry: './src/file1.js',

  output: {

    filename: 'bundle.js',

    path: path.resolve(__dirname, 'dist'),

  },

};
```

Step 4: Run Webpack:

```
npx webpack --config webpack.config.js
```

Explanation:

- In this example, `file1.js` serves as the entry point specified in the Webpack configuration. Webpack will analyze the dependencies & generate a bundled file named `bundle.js`.

- The `webpack.config.js` file defines the entry point (`./src/file1.js`) & the output file (`bundle.js` in the `dist` folder).

- After running Webpack, you will get a bundled file (`bundle.js`) that includes the content of both `file1.js` & `file2.js`.

5. What is the process of first app in node.js.

Ans. Node.js is an open source, cross-platform server environment which executes JavaScript using V8 JavaScript Engine. Node.js helps to write front-end & back-end code in the same language. It helps to write efficient code for real-time applications. In Node.js, the applications can be written using console-based method or web-based method.

Syntax: console.log([data][, ...]);

Creating Your First Node.js Application:

Step 1: Import Required Module

In Node.js, modules are used to organize code into separate files. The `http` module is a core module that provides functionality to create an HTTP server. We use the `require` directive to load this module.

// main.js

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

Step 2: Create Server

To create an HTTP server, we utilize the `http.createServer()` method. This method takes a callback function that will be invoked whenever a request is made to the server. In this callback, we h&le the response by setting the HTTP header & sending the "Hello World" message. The `listen` method binds the server to a specific port, in this case, 8081.

```
// main.js

const http = require("http");

const server = http.createServer((request, response) => {

  // Send the HTTP header

  // HTTP Status: 200 OK

  // Content Type: text/plain

  response.writeHead(200, {'Content-Type': 'text/plain'});

  // Send the response body as "Hello World"

  response.end('Hello World\n');

});

server.listen(8081);

console.log('Server running at http://127.0.0.1:8081/');
```

Step 3: Testing Request & Response

The server is now ready to h&le requests. When the server receives a request, it responds with "Hello World" in plain text. Execute the following comm& to start the server.

```
$ node main.js
```

Verify the Output:

```
Server running at http://127.0.0.1:8081/
```

Make a Request to the Node.js Server:

Open http://127.0.0.1:8081/ in any browser to observe the result.

Underst&ing the Code:

- We import the `http` module to access its functionality.

- The `createServer` method creates an HTTP server instance with a callback function to h&le requests.

- The server listens on port 8081, & the console displays a message indicating the server is running.

- When you access http://127.0.0.1:8081/ in a browser, the server responds with "Hello World."