Unit 2: HTML5, JQuery and Ajax

HTML5 <audio> tag

It is used to play audio in html pages. It takes the below basic format in its simplest form:

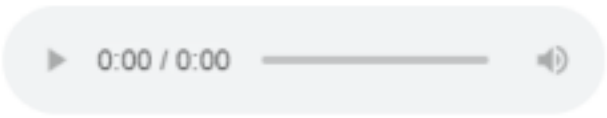
<audio src="my_music.mp3" controls></audio>

With the above structure, when the html page loads the page requests for the audio file listed in the "src" attribute and the “controls” attribute displays the browser default audio player for controlling playback.

CODE:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Intro to Audio</title>
</head>
<body>
    <audio src="sample.mp3" controls></audio>
</body>
</html>
```

Results:



The <audio> tag comes with many inline global attributes that help in modifying its behaviour. Some of the attributes are:

1. autoplay
2. buffered
3. controls
4. loop
5. muted
6. played
7. preload
8. src
9. Volume

Some of the attributes, values and description are given below:

Attribute	Value	Description
autoplay	autoplay	Specifies that the audio will start playing as soon as it is ready.
controls	controls	Specifies that controls will be displayed, such as a play button.
loop	loop	Specifies that the audio will start playing again (looping) when it reaches the end
preload	preload	Specifies that the audio will be loaded at page load, and ready to run. Ignored if autoplay is present.
src	*url*	Specifies the URL of the audio to play

HTML 5 <video> Tag

The HTML 5 <video> tag is used to specify video on an HTML document. For example, it can be embed in a music video on web page for visitors to listen to

and watch. The <video> tag was introduced in HTML 5.

The HTML 5 <video> tag accepts attributes that specify how the video should be played. Attributes include preload, autoplay, loop and more.

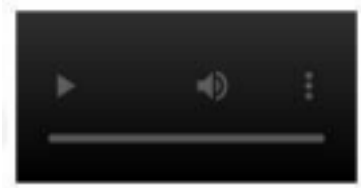
Any content between the opening and closing <video> tags is fallback content. This content is displayed only by browsers that don't support the <video> tag.

Attribute	Value	Description
audio	muted	Defining the default state of the the audio. Currently, only "muted" is allowed
autoplay	autoplay	If present, then the video will start playing as soon as it is ready
controls	controls	If present, controls will be displayed, such as a play button
height	*pixels*	Sets the height of the video player
loop	loop	If present, the video will start over again, every time it is finished
poster	*url*	Specifies the URL of an image representing the video
preload	preload	If present, the video will be loaded at page load, and ready to run. Ignored if "autoplay" is present
src	*url*	The URL of the video to play
width	*pixels*	Sets the width of the video player

CODE:

```
<video src="/video/pass-countdown.ogg" width="170" height="85" controls>
<p>If you are reading this, it is because your browser does not support the HTML5 video element.</p>
</video>
```

Results:



HTML 5 <progress> Tag

The <progress> element is used to create a progress bar to serve as a visual demonstration of progress towards the completion of task or goal. The max and value attributes are used to define how much progress (value) has been made towards task completion (max).

It can be indeterminate progress bar, which can be either in the form of spinning wheel or a horizontal bar. In this mode, the bar only shows cyclic movements and do not provide the exact progress indication. This mode is usually used at the time when the length of the time is not known.

CODE: *Results:*

```
<progress value="33" max="100"></progress>
```

Reference Link for styling progress elements:

https://css-tricks.com/html5-progress-element/

Unit 2: HTML5, JQuery and Ajax

HTML5 Geolocation API:

The Geolocation API of HTML5 helps in identifying the user's location, which can be used to provide location specific information or route navigation details to the use. There are many techniques used to identify the location of the user. The Geolocation API protects the user's privacy by mandating that the user permission should be sought and obtained before sending the location information of the user to any website. So the user will be prompted with a popover or dialog requesting for the user's permission to share the location information. The user can accept or deny the request.

The current location of the user can be obtained using the getCurrentPosition function of the navigator.geolocation object. This function accepts three parameters – Success callback function, Error callback function and position options. If the location data is fetched successfully, the success callback function will be invoked with the obtained position object as its input parameter. Otherwise, the error callback function will be invoked with the error object as its input parameter.

Example: This example explains returning the user's current location using the getCurrentPosition() method.

```
<!DOCTYPE html>
<html>
  <body>
  <p>Displaying location using Latitude and Longitude</p>  <button
class="geeks" onclick="getlocation()"> Click Me </button>
  <p id="demo1"></p>
  <script>
  let variable1 = document.getElementById("demo1");  function
```

```
getlocation() {
  navigator.geolocation.getCurrentPosition(showLoc);  }
  function showLoc(pos) {
  variable1.innerHTML = "Latitude: " + pos.coords.latitude  +
"<br>Longitude: " + pos.coords.longitude;
  }
  </script>
  </body>
</html>
```

Unit 2: Callbacks and Promises

Callbacks in JavaScript

- **Definition**

  A callback is a function provided as an argument to another function, which gets executed after the main task is done.

- **Why Callbacks?**

  - JavaScript is synchronous by default—it executes lines one by one.
  - When a function involves time-consuming tasks (like animations, timers, or network requests), the rest of the code may keep executing before that task finishes.
  - Callbacks allow you to specify code that runs **after** the asynchronous task is complete, ensuring correct order.

- **Callback Hell**

  - If multiple asynchronous tasks depend on each other, callbacks get deeply nested, messy, and hard to maintain ("callback hell").

Promises in JavaScript

- **Definition**

A Promise is an object returned by asynchronous functions, representing a value that may be available now, later, or never (if the operation fails).

- **States of a Promise**
  - **Pending:** Operation is still ongoing.
  - **Fulfilled:** Operation completed successfully.
  - **Rejected:** Operation failed with an error.

- **How to Create and Use a Promise**

```
const myPromise = new Promise(function(resolve, reject) {
      // ... do some async work
      if (/* success */) {
      resolve(result);
      } else {
      reject(error);
      }
     });

     myPromise
      .then(function(result) {
      // Runs if resolved: handle the result
      })
      .catch(function(error) {
      // Runs if rejected: handle the error
      });
```

- **Example: Get Author's Name After an Async Task**

```
// Old way (synchronous):
var author = getAuthors();
var authorName = author.name;

// With Promises (asynchronous):
getAuthors().then(function(author) {
 return author.name;
});
```

- **Another Example: Weather Promise**

```
const weather = true;
const date = new Promise(function(resolve, reject) {
 if (weather) {
 const dateDetails = {
 name: 'Cubana Restaurant',
 location: '55th Street',
 table: 5
 };
resolve(dateDetails);
 } else {
 reject(new Error('Bad weather'));
 }
});

date
 .then(function(done) {
 console.log('We are going on a date!');
 console.log(done);
```

```
})
.catch(function(error) {
console.log(error.message);
});
```

- If `weather` is true, prints the details of the date.
- If `weather` is false, prints "Bad weather".

- **Chaining and Handling**
  - Use `.then()` for success; use `.catch()` for errors.
  - This avoids callback hell and makes code easier to read and maintain.

Summary Table

Concept	What It Means
Callback	Function executed after async task completes
Callback Hell	Complex, unreadable code from too many nested callbacks
Promise	Object representing eventual success/error of async task
States	pending, fulfilled, rejected
.then()	Handles fulfilled (success) state
.catch()	Handles rejected (error) state

Key Takeaways

- Use **callbacks** for simple async logic; be careful not to nest too deeply!
- **Promises** are the modern, clean way to handle multiple async operations.
- Always use `.then()` and `.catch()` to handle the different outcomes of a Promise.

Unit 2: HTML5, JQuery and Ajax
XML Vs JSON

XML Vs JSON

JSON, XML, are Text-file formats that can be used to store structured data that can be handy for embedded and Web applications.

XML (Extensible Markup Language) has been around for more than 3 decades now and it is an integral part of every web application. Be it a configuration file, mapping document or a schema definition, XML made life easier for data interchange by giving a clear structure to data and helping in dynamic configuration and loading of variables!

JSON stores all of its data in a map format (key/value pairs) that was neat and easier to comprehend. JSON is said to be slowly replacing XML because of several benefits like ease of data modeling or mapping directly to domain objects, more predictability and easy to understand the structure. JSON is just a data format whereas XML is a markup language.

Structure of XML vs JSON :

XML (Extensible Markup Language)	JSON (JavaScript Object Notation)

01
Adam
Cloud computing </technology >
 	
Development
	{
	"employees":
	[
	{
	"id": "01",
	"name": "Adam",
	"technology": "Cloud computing",
	"title": "Engineer",
	"team": "Development"
	}
	}

Difference between XML and JSON:

XML (Extensible Markup Language)	JSON (JavaScript Object Notation)
XML is a markup language, not a programming language, that has tags to define elements.	JSON is just a format written in JavaScript.
XML data is stored as a tree structure. Example – 2001 Varsha 2002 Akash	Data is stored like a map with key value pairs. Example – {“employees”: [{"id":"2001", “name":"Varsha"}, {"id":"2002", "name":"Akash"}]}

Can perform processing and formatting documents and objects.	It does not do any processing or computation
Bulky and slow in parsing, leading to slower data transmission	Very fast as the size of file is considerably small, faster parsing by the JavaScript engine and hence faster transfer of data
Supports namespaces, comments and metadata	There is no provision for namespace, adding comments or writing metadata
Document size is bulky and with big files, the tag structure makes it huge and complex to read.	Compact and easy to read, no redundant or empty tags or data, making the file look simple.
Doesn't support array directly. To be able to use array, one has to add tags for each item. science maths computers	Supports array which can be accessed as – x = student.subjects[i]; **where** "subjects" **is** an array **as** – "subjects": ["science", "math", "computers"]
Supports many complex data types including charts, images and other non-primitive data types.	JSON supports only strings, numbers, arrays Boolean and object. Even object can only contain primitive types.
XML supports UTF-8 and UTF-16 encodings.	JSON supports UTF as well as ASCII encodings.

XML structures are prone to some attacks as external entity expansion and DTD validation are enabled by default. When these are disabled, XML parsers are safer.	JSON parsing is safe almost all the time except if JSONP is used, which can lead to Cross-Site Request Forgery (CSRF) attack.
Though the X is AJAX stands for XML, because of the tags in XML, a lot of bandwidth is unnecessarily consumed, making AJAX requests slow.	As data is serially processed in JSON, using it with AJAX ensures faster processing and hence preferable. Data can be easily manipulated using eval() method.

XML Parser

The XML DOM (Document Object Model) defines the properties and methods for accessing and editing XML.

XML parser is a software library or a package that provides interface for client applications to work with XML documents. It checks for proper format of the XML document and may also validate the XML documents.

XMLSerializer

The XMLSerializer interface provides the serializeToString() method to construct an XML string representing a DOM tree.

Eg –

```
var s = new XMLSerializer();
 var d = document;
 var str = s.serializeToString(d);
 saveXML(str);
```

Example

```
var text =
"<studentdetails><student>"+"<name>John</name>"+"<subject>Web</subj
ect>"+"</studentdetails></student>";

//converts a DOM string to XML DOM structure

var parser = new DOMParser();

var xmldoc = parser.parseFromString(text,"text/xml");

document.getElementById("demo").innerHTML =

xmldoc.getElementsByTagName("name")[0].childNodes[0].nodeValue;
```

JSON

- JSON: JavaScript Object Notation.
- JSON is a syntax for storing and exchanging data.
- JSON is text, written with JavaScript object notation.
- JSON is a lightweight data-interchange format
- JSON is "self-describing" and easy to understand
- JSON is language independent

JSON uses JavaScript syntax, but the JSON format is text only. Text can be read and used as a data format by any programming language.

JSON NOTATION

JSON data is written as name/value pairs.

A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value. The values can be a string, number, an object(JSON object),an array, a Boolean, null.

Eg-

{ "name": "John" } in JSON and

{ name: "John" } in JavaScript

If you have data stored in a JavaScript object, you can convert the object into JSON, and send it to a server:

```
var obj = { name: "John", age: 30, city: "New York" };

var myJSON = JSON.stringify(obj);

document.getElementById("demo").innerHTML = myJSON;
```

If you receive data in JSON format, you can convert it into a JavaScript object:

```
var myJSON = '{ "name":"John", "age":31, "city":"New York" }';

var myObj = JSON.parse(myJSON);

document.getElementById("demo").innerHTML = myObj.name;
```

JSON vs XML formats

Eg- employee details

```
{"employees":[
  { "firstName":"John", "lastName":"Doe" },
  { "firstName":"Anna", "lastName":"Smith" },
  { "firstName":"Peter", "lastName":"Jones" }
]}
```


```
<employees>
  <employee>
    <firstName>John</firstName> <lastName>Doe</lastName>
  </employee>
  <employee>
    <firstName>Anna</firstName> <lastName>Smith</lastName>
  </employee>
  <employee>
    <firstName>Peter</firstName> <lastName>Jones</lastName>
  </employee>
</employees>
```

1. What is an SPA?

- **Definition**:
  A **Single Page Application (SPA)** loads a **single HTML page** and dynamically updates its content as the user interacts, instead of performing a full page reload.
- **How it feels**: More like a **native mobile app** than a traditional web page.
- **Examples**: Gmail, Google Maps, Trello, Netflix, Flipkart Web App.
- **Why important**: Provides **fast, smooth user experience** and is the standard for most modern web apps.
- **Analogy**: Think of Instagram — when you scroll, **new posts load instantly**, but the app never reloads.

Extra Info:

- SPAs rely heavily on **JavaScript frameworks** (React, Angular, Vue) to handle rendering and state.
- They usually communicate with the server via APIs (REST, GraphQL).

---

2. How SPAs Work

- **Initial Load**:
  - Browser loads **HTML, CSS, JS** in one go (often a big bundle).
- **Subsequent Interactions**:
  - Use **AJAX, Fetch API, or GraphQL queries** to request data.
  - The **DOM is partially updated** instead of reloading the whole page.
- **Client-side Routing**:
  - Handled with history.pushState or frameworks like **React Router**, **Angular Router**.
  - Allows different “views” without actual new HTML pages.

- **Result**: Seamless, fast transitions with an “app-like” feel.

Extra Info:

- Modern SPAs use **hydration** (server sends pre-rendered HTML, then JS takes over) to improve SEO and load times.
- Service workers can cache assets, enabling **offline support (PWA)**.

---

3. Why SPAs are Awesome

- **Speed**:
  - After the initial load, pages update instantly → less bandwidth, snappy feel.
- **App-like UX**:
  - Smooth transitions, no flicker, feels like using a mobile app.
- **Separation of Concerns**:
  - Backend = APIs, Frontend = rendering/UI.
- **Developer Benefits**:
  - Easier to build modular, reusable components.
  - Shared frontend logic between web & mobile (e.g., React + React Native).

Extra Info:

- SPAs are the **default approach** for startups and SaaS dashboards.
- Popular because of **reduced server rendering costs** and **fast iteration cycles**.

Fun Note: SPAs = “YouTube at 2x speed” → same content, faster experience.

---

4. SPA Challenges (The Flip Side)

- **SEO Limitations**:

- Content rendered via JS is hard for crawlers to index.
- Solutions: **SSR (Next.js, Nuxt.js)**, **pre-rendering**, or using **dynamic rendering services** like Rendertron.

- **Routing / History Handling**:
  - Browser back/forward buttons don't work by default.
  - Requires libraries (React Router, Angular Router).
- **Performance**:
  - Initial load can be heavy (large JS bundles).
  - Code splitting + lazy loading helps.
- **Security**:
  - Heavy client-side code increases risks (XSS, data exposure).
  - Must sanitize user inputs and secure APIs.

Extra Info:

- Large SPAs risk **memory leaks** if event listeners and components aren't cleaned up properly.
- SPAs can struggle on **low-end devices** if bundles are too large.

Analogy: A Tesla = smooth and modern, but needs careful setup (charging, maintenance).

---

5. Multi-Page Applications (MPA)

- **Definition**:
  - Each new interaction loads a **brand-new HTML page** from the server.
- **Workflow**:

1. User clicks a link → request sent to server.
2. Server processes and sends back a new HTML page.

3. Browser replaces the old page.

- **Examples**: Old IRCTC, SBI Netbanking, blogs, news sites.
- **Strengths**:
  - SEO-friendly (crawler-friendly HTML).
  - Easier server-side security.
  - Suitable for content-heavy sites.
- **Limitations**:
  - Slower UX (every click = reload).
  - Higher bandwidth.
  - Navigation feels "choppy."

Analogy: Waiter bringing you a **new plate every time** you order, vs SPA where your plate just gets refilled.

---

6. SPA vs MPA (Comparison)

Feature	SPA	MPA
Page Reloads	None after first load	Every navigation
Speed/UX	Seamless, app-like	Choppy, reloads often
SEO	Needs extra setup (SSR)	Works naturally
Development Model	API-driven frontend	Server renders HTML
Best Use Case	Interactive apps (Gmail, Trello, Netflix)	Content-heavy sites (blogs, news)

Extra Info:

- Hybrid approaches exist → e.g., **Next.js** (SPA + SSR) offers the best of both worlds.

7. Common Use Cases of SPAs

- **Ideal For**:
  - Email apps (Gmail, Outlook).
  - Streaming platforms (Netflix, Hotstar).
  - Dashboards & data-heavy apps.
  - Collaboration tools (Slack, Trello, Jira).
- **Not Great For**:
  - SEO-heavy content websites (news portals, product catalogs).
- **Mnemonic**:
  - **Interactive = SPA**
  - **Static info = MPA**

8. Best Practices & SPA Hygiene

- **Performance**:
  - Use **code splitting, lazy loading, tree shaking**.
  - Compress and cache assets.
- **SEO**:
  - Implement SSR (Next.js, Nuxt.js).
  - Pre-render static content where possible.
- **Routing**:
  - Use proper libraries for navigation.
- **Caching & Offline**:
  - Service Workers → offline-first, PWA support.

- **Security**:
  - Sanitize inputs, secure APIs, store sensitive logic server-side.
- **Developer Tip**:
  - Always test SPAs under **realistic conditions** (slow 3G/4G).

---

9. Summary

- SPAs = modern, smooth, app-like experience.
- MPAs = traditional, secure, SEO-friendly but slower.
- SPAs shine in interactive apps, MPAs in static/content-rich apps.
- SPA best practices include performance optimization, SEO handling, and secure coding.

Unit 2: Async and Await in JavaScript

1. Async Programming

- JavaScript runs code single-threaded. Blocking operations (like network requests) can freeze the app until complete.
  - **Asynchronous programming** lets code run in the background, keeping interfaces fast and smooth.

2. Promises

- Promises represent values that may be available now, later, or never.
- Syntax:

```
let promise = new Promise((resolve, reject) => {
 // do something async
});
```

- You handle outcomes with `.then()` and `.catch()`.

3. async and await

- **async**: Declares a function as asynchronous. It **always returns a Promise**—even if your code inside returns a value (which will be wrapped in a resolved Promise).
- **await**: Pauses execution in an `async` function until a Promise settles. Returns the resolved value, or throws if the Promise is rejected.
- **Goal:** Make asynchronous code read and behave more like synchronous code.

4. Syntax and Basic Usage

A. Declaring an async Function

```
async function greet() {
 return "Hello, Async!";
}
greet().then(msg => console.log(msg)); // Prints: Hello, Async!
```

B. Using Await Inside async Function

```
async function waitAndPrint() {
 await new Promise(res => setTimeout(res, 1000));
 console.log("Finished waiting!");
}
waitAndPrint();
```

5. How await Works

- Only works inside `async` functions.
- Pauses execution of function at that point until the awaited promise resolves/rejects. •

Lets you write flat, readable sequences instead of chains.

6. Practical Example: Fetch Data with Error Handling

```
async function fetchUser(userId) {
 try {
 const response = await
fetch(`https://jsonplaceholder.typicode.com/users/${userId}`)
; const user = await response.json();
 console.log(user);
 } catch (error) {
 console.error("Failed to fetch user:", error);
 }
}

fetchUser(1);
```

7. Sequential vs Parallel await

A. Sequential

Do one after another, each depends on the previous finish:

```
async function getUserAndPosts(userId) {
 const userResponse = await fetch(`.../users/${userId}`);
const user = await userResponse.json();
 const postsResponse = await fetch(`.../posts?userId=${userId}`);
const posts = await postsResponse.json();
 return [user, posts];
}
```

Use case: When the second request needs data from the first.

B. Parallel (with Promise.all)

Start at the same time for better speed!

```
async function getMany() {
 const [user, post] = await Promise.all([
 fetch('.../users/1').then(res => res.json()),
 fetch('.../posts/1').then(res => res.json())
 ]);
 console.log(user, post);
}
```

Use case: When operations are independent.

8. Simulating Delays

```
function wait(ms) {
return new Promise(resolve => setTimeout(resolve, ms));
}
async function demoDelay() {
console.log('Waiting...');
await wait(2000);
console.log('2 seconds later!');
}
demoDelay();
// Output: Waiting... [2 second pause] 2 seconds later!
```

9. Error Handling with try/catch

• Place `await` calls in a `try` block. If any throws/rejects, `catch` runs.

• Cleaner than repeatedly attaching `.catch()` to each promise.

10. Best Practices

• **Always use `try/catch`** around awaits in production code.

• Use Promise.all for true parallelism when tasks are independent.

• Avoid blocking heavy computation with await—suitable only for IO/network. • Remember, every async function returns a Promise!

• Use top-level await only in ES Modules (`.mjs` files or `<script type="module">` in HTML).

11. Common Real-world Use Cases

- Fetching REST API data in web apps.
- Database or file operations in Node.js.
- Series of network calls where later depend on the earlier.
- Waiting for animations or user actions to complete (with Promises).

12. Extra Practical Examples

Looping with async/await (Serial Execution)

```
async function printNumbers() {
 for (let i = 1; i <= 3; i++) {
 await wait(500);
 console.log(i);
 }
}
printNumbers();
// Prints 1, 2, 3 at 0.5 second intervals
```

Using async function as a callback (for event handlers etc.)

```
button.addEventListener('click', async () => {
 try {
 await someAsyncTask();
 alert('Task done!');
 } catch (e) {
 alert('Failed: ' + e.message);
 }
});
```

13. Comparison Table

Feature	Callback	Promise Async/Await
Syntax	Nested	Chained Flat/readable
Error Handling	Callback arg	.catch() try/catch
Readability	Low	Medium High
Parallelism	Hard	Possible Easy with Promise.all

14. Key Takeaways

- `async` and `await` make asynchronous code easier and safer.
- They are standard in all modern JavaScript environments—learn and use them!
- Handle errors robustly and always await Promises for correctness.

Unit 2: HTML5, JQuery and Ajax

AXIOS

1. Axios?

- **Axios** is a promise-based JavaScript library for making HTTP requests from both browsers and Node.js.
- It is designed to interact easily with APIs, especially RESTful backends.
- Axios automatically transforms JSON data and supports modern async JavaScript patterns.

2. Key Features

- **Promise-based API:** Works seamlessly with `.then()`, `.catch()`, or `async/await`. •

Cross-environment: Runs in browsers and Node.js with the same code.

- **Automatic JSON transformation:** Converts JavaScript objects to JSON in requests and parses JSON in responses.
- **Request/Response Interceptors:** Easily add logic before requests leave your app or after responses come in.
- **Easy error handling:** Built-in handling for network/server errors.
- **Supports all HTTP methods:** GET, POST, PUT, DELETE, PATCH, etc.

- **Request cancellation and timeout support.**

3. Installation

In Node.js Projects:

```
npm install axios
```

Import in your JS file:

```
const axios = require('axios');
// or for ES6:
import axios from 'axios';
```

In Browsers (via CDN):

```
<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
```

4. Basic Usage Examples

GET Request:

```
axios.get('https://jsonplaceholder.typicode.com/posts/1')
  .then(response => {
 console.log(response.data);
```

```
})
.catch(error => {
console.error('GET error:', error);
});
```

POST Request:

```
axios.post('https://jsonplaceholder.typicode.com/posts', {
 title: 'My Post',
 body: 'This is the post content.',
 userId: 1
})
 .then(response => {
 console.log('Created:', response.data);
})
 .catch(error => {
 console.error('POST error:', error);
});
```

5. Advanced Request Options

Common Options

- **Headers:** Add custom headers like auth tokens, content types.
- **Params:** Set query parameters easily.
- **Timeout:** Abort requests that take too long.
- **Response Type:** Specify expected response format (`json`, `blob`, etc.).

Example:

```
axios.get('https://api.example.com/search', {
 params: { q: 'axios', limit: 5 },
 headers: { 'X-API-KEY': 'abc123' },
 timeout: 3000, // 3 seconds
 responseType: 'json'
})
 .then(res => console.log(res.data))
 .catch(err => console.error('Error:', err));
```

6. Handling Responses

- **response.data:** The main content/body returned by the server.
- **response.status:** The HTTP status code (e.g., 200, 404).
- **response.headers:** The response headers object.
- **response.config:** The request configuration.

7. Global Configuration

Set defaults so you don't repeat yourself in every request:

```
axios.defaults.baseURL = 'https://api.example.com';
axios.defaults.headers.common['Authorization'] = 'Bearer TOKEN123';
axios.defaults.timeout = 5000;
```

Or, create a custom instance:

```
const customAxios = axios.create({
 baseURL: 'https://api.example.com',
 timeout: 2000,
});
```

8. Interceptors

Request Interceptor: Add a token or log every request:

```
axios.interceptors.request.use(config => {
 config.headers['Authorization'] = 'Bearer ' + getToken();
 return config;
}, error => Promise.reject(error));
```

Response Interceptor: Handle errors globally:

```
axios.interceptors.response.use(
 response => response,
 error => {
 if (error.response.status === 401) {
 // Redirect to login, for example
 }
 return Promise.reject(error);
 }
);
```

10. Practical Best Practices

- **Always handle errors** using `.catch()` or `try-catch` blocks.
- **Validate response data** before using it in your UI.
- **Avoid exposing secrets** (tokens, passwords) in frontend code.
- Prefer **async/await** for cleaner, flatter code.
- Use **interceptors** for repetitive logic (like adding tokens, logging).
- **Cancel requests** if a user navigates away or data is no longer needed.

11. Real-world Use Cases

- Consuming public REST APIs (e.g., weather, news, product data).
- Authenticated requests using JWT in React/Vue/Angular apps.
- Server-side rendering with Node.js (fetch data before serving HTML).
- Uploading/downloading files (using `FormData` and responseType options).

12. Comparison: Axios vs Fetch

Feature	Axios	Fetch API
JSON auto-transform	Yes	No (manual)
Interceptors	Yes	No
Older browser support	Yes	Polyfill needed
Request cancellation	Yes	Yes (newer only)
Error handling	On non-2xx triggers	Only on network fail
Upload/download files	Easy with FormData	Manual config

13. Reference Links

- Axios Official Documentation
- MDN: Using Promises
- Axios GitHub Repository**

UNIT 2 - ReactJS - MERN Introduction

Web Development Stack

- A **web development stack** combines frontend, backend, and database technologies to build modern web applications.
- Analogy: Like a restaurant—frontend is the menu/dining area (user interface), backend is the kitchen (business logic), and database is the pantry (data storage).
- A **full stack** handles all three layers, e.g., MERN (MongoDB, Express, React, Node.js).

MERN Stack Introduction

- MERN stands for **MongoDB, Express.js, React.js, and Node.js**—all JavaScript-based technologies.
- MongoDB: NoSQL document database (the pantry).
- Express.js: Lightweight server framework for Node.js (chef assistant).
- React.js: Client-side UI library (waiter presenting data).
- Node.js: JavaScript runtime environment on server (the kitchen itself).
- Using the same language (JavaScript) on frontend and backend improves development efficiency.

ReactJS Overview

- React is a **component-based JavaScript library** for building dynamic user interfaces.
- Uses a **Virtual DOM** for efficient updates, separating the UI into reusable components.
- React supports **single-way data flow**: data passes from parent to child components; actions from child propagate back up through callbacks.
- Popular for building **single-page applications (SPAs)** with smooth and fast user experience.

Single-Way Data Flow in React

- Data flows down from **parent** → **child** via **props** (read-only).
- Child components update the parent or global state by invoking **callbacks** passed down as props (actions flow up).
- This unidirectional flow makes data handling more predictable and debugging easier.
- Analogy: Like a waterfall, water flows only downstream, but signals can go upstream.

Virtual DOM (VDOM)

- The **Virtual DOM** is a lightweight copy of the real DOM held in memory.
- React creates a VDOM tree when rendering and generates a new tree when state/props change.

- React uses a **diffing algorithm** to compare old vs new VDOM, and only updates the real DOM nodes that changed.
- This dramatically improves performance by avoiding unnecessary DOM manipulation.
- Analogy: Like sketching room changes on paper before rearranging furniture physically.

Creating a React App

- **Method 1:** Using npm and Create React App:

```
npx create-react-app my-app
cd my-app
npm start
```

Quick React Properties Overview

- **Declarative:** Describe what UI should look like; React figures out how to update.
- **Simple:** Focuses on readability and maintainability.
- **Component-Based:** UI built from reusable components.
- **Server-Side Rendering:** Supported for SEO and performance.
- **Mobile Support:** Through React Native.
- **Fast:** Uses Virtual DOM for efficient updates.
- **Single-Way Data Flow:** Props down, events up.

UNIT 2 - JSX, rendering of Elements

JSX – Rendering of Elements

- **JSX** stands for **JavaScript XML**; it's a syntax extension that allows writing HTML-like code inside JavaScript.
- Enables easier creation of UI components by mixing markup with logic.
- Though it looks like HTML, **JSX is transformed into JavaScript function calls** (`React.createElement`) by tools like Babel.
- These transformed objects (React elements) are used by React to update the actual DOM efficiently.

Converting HTML to JSX

- When using JSX inside React components, **HTML must be converted properly to JSX syntax**.
- Common rules for conversion:

1. **Single Root Element**: JSX must return one enclosing parent element. Use `<div>`, `<section>`, or React Fragment syntax `<></>` to wrap siblings.
2. **Close All Tags**: Self-closing tags like `<img>`, `<input>` must include a trailing slash: `<img />`.
3. **Use camelCase Attributes**: HTML attributes are named differently in JSX. For example:

- `class` → `className`
- `for` → `htmlFor`
- `tabindex` → `tabIndex`

Embedding Expressions in JSX

• JavaScript expressions can be embedded inside JSX using curly braces `{}`. • Examples include variables, function calls, conditional expressions, and array methods. • JSX expressions are compiled into JavaScript function calls returning React elements.

Examples of JSX Embedding

• Variable embedding:

```
const name = 'Josh Perez';
const element = <h1>Hello, {name}</h1>;
```

• Conditional rendering:

```
const isLoggedIn = true;
const message = <h1>{isLoggedIn ? 'Logout' : 'Login'}</h1>;
```

- Rendering lists with keys:

```
const numbers = [1, 2, 3];
const listItems = numbers.map(num => <li
key={num.toString()}>{num}</li>);
const element = <ul>{listItems}</ul>;
```

Rendering JSX to DOM (React 18+)

- Use `ReactDOM.createRoot` to create a root and `root.render()` to render JSX to the DOM.
- Example:

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<h1>Hello React!</h1>);
```

JSX Spread Attributes

- Spread operator `...` can be used to pass multiple props to components or elements concisely.

- Example:

```
const props = { multiple: true, disabled: false };
<input type="checkbox" {...props} />;
```

Handling Events in JSX

- Event handler attribute names use camelCase, e.g., `onClick`, `onChange`.
- Pass event handler functions, not strings.
- Example:

```
<button onClick={() => alert('Clicked!')}>Click Me</button>
```

Controlled Components in JSX Forms

- Form input value is controlled by React state.
- Update state through `onChange` event handlers to reflect user input.

- Example:

```
function MyForm() {
  const [name, setName] = React.useState('');
  return <input type="text" value={name} onChange={e =>
setName(e.target.value)} />;
}
```

JSX Comments

- Comments inside JSX need to be enclosed in `{/* comment */}` syntax, not HTML comment syntax.

Best Practices

- Always provide **unique keys** when rendering lists to minimize unnecessary re-renders.
- Avoid anonymous functions inline for event handlers if possible to improve performance.
- Break UI into smaller reusable components to keep code organized and maintainable.

Example: Conditional Rendering with LogicalAND

- Logical AND operator short-circuits rendering:

```
const show = false;
const element = <div>{show && <p>Hello, World!</p>}</div>;
```

- Since `show` is false, the `<p>` element is not rendered, resulting in an empty `<div>`.

React Setup & Installation (npm)

What is npm?

- **NPM (Node Package Manager)** is the default package manager for **Node.js**.
- Used to:
    - Manage **project dependencies** (third-party libraries).
    - Run **scripts** (e.g., start server, build app).
    - Share and reuse **packages/modules**.
- Installing Node.js also installs npm automatically.

A **package** = a reusable module of code (from a small utility to a full-featured library).
Installed packages are stored in the **node_modules** folder.

Installing Node.js & npm

1. Download & install Node.js from:
   Node.js Installation Guide
2. During installation → Add Node.js to **PATH**.
3. Verify installation:
    1. node -v
    2. npm -v

Setting up a React Project

npm create vite@latest my-react-app -- --template react

cd my-react-app

npm install (Install all dependencies)

npm run dev

React Functional Components & JSX

What are Functional Components?

- Functional Components = **JavaScript functions** that return **JSX** (JavaScript XML).
- They are the **building blocks of modern React apps**.
- Syntax:

```
function Welcome() {
  return <h1>Hello, World!</h1>;
}

// Arrow function
const Welcome = () => <h1>Hello, World!</h1>;
```

- JSX must return a **single root element**.

Why Use Functional Components?

- Promote **code reusability** (write once, use anywhere).
- Help maintain **consistency** in UI.
- Easy to maintain and test.
- Default are **stateless**, but can use **React Hooks** for state & side effects.

JSX (JavaScript XML)

- Looks like **HTML inside JavaScript**.
- Must be enclosed in **one root element**.
- Allows embedding **expressions** inside { }.

```
function App() {
  return (
    <div>
      <h1>Hello JSX!</h1>
      <p>2 + 2 = {2 + 2}</p>
    </div>
  );
}
```

Props (Properties)

- **Props** = way to **pass data** into components.

- Passed like HTML attributes.
- **Immutable** (cannot be modified inside the component).
- Example:

```
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}

<Greeting name="Alice" />
```

UNIT 2 - React Component Styling

React Component Styling: Overview & Importance

- Styling in React applies visual design rules to individual components. Important for improving user experience and code maintainability.
- React promotes **component-scoped styling**, reducing conflicts and enhancing modularity.
- Choosing the right styling strategy optimizes performance and collaboration.

Traditional Styling with CSS Files

- Use external `.css` files or **CSS Modules** for scoped styling (e.g., `Button.module.css`).
- Import styles into components and apply via `className={styles.primary}`.
- Encourages separation of concerns and keeps styles organized.
- Analogy: Like school uniforms providing consistent, conflict-free dressing.

Inline Styling – Quick but Limited

- Use inline style objects in JSX:

```
<div style={{ color: 'red', fontSize: '20px' }}>Hello Style!</div>
```

- Good for quick, dynamic styles but harder to maintain.
- Lacks support for CSS features like pseudo-classes (`:hover`).
- Analogy: Like requesting extra spices on a single dish – fast but not scalable.

CSS-in-JS (Styled-Components & Others)

- Define styles directly within JavaScript using libraries like **styled-components**, **Emotion**, or **JSS**.
- Provides encapsulation, theming, and eliminates style collisions.
- Styles live alongside component logic, improving cohesion.
- Analogy: Like grocery delivery bundling spices and groceries together in one package.

Styling Utilities & UI Frameworks

- **Utility-first CSS frameworks** like **TailwindCSS** allow composing styles through classes

(e.g., `bg-blue-500`, `text-white`).

- **Component Libraries** (Material UI, Chakra UI, Ant Design) offer ready-made, themeable components.
- Analogy: Utility-first = stacking filters on Zomato; Component libraries = ordering a customized pizza from Domino's.

Modern Best Practices

- Combine **Hooks** with **CSS-in-JS** for modular, feature-aligned styling.
- Prefer CSS Modules or CSS-in-JS for isolation and maintainability.
- Scope styles per component for reuse and clarity.
- Analogy: Like individual Netflix profiles keeping content personalized yet organized under one subscription.

React 19 & `<style>` Handling

- React optimizes `<style>` tags: moves them to `<head>`, removes duplicates, respects precedence.
- Results in better performance and fewer CSS conflicts.

Choosing the Right Styling Strategy

Scenario	Best Option	Why This Works
Simple UI, small projects	CSS	overhead Good
Large apps needing themes	CSS-in-JS (styled-components)	encapsulation, easy theming
CSS Modules or traditional	Quick setup, minimal	
Utility-first fans	Tailwind + UI Libraries	Fast, consistent styling, flexible
Enterprise-scale apps	UI Frameworks (MUI, Chakra)	Accessibility, polish, ready-to-use components