Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

ADVANCED JAVASCRIPT & JQUERY (WEEKS 13-15)

WORKING WITH ARRAYS & OBJECTS IN JAVASCRIPT

CHAPTER 1: INTRODUCTION TO ARRAYS & OBJECTS IN JAVASCRIPT

1.1 What Are Arrays & Objects?

In JavaScript, **arrays and objects** are used to store and manipulate collections of data.

- **Arrays** → Used to store multiple values in a single variable.
- **Objects** → Used to store **key-value pairs**.
- **Example:**

let fruits = ["Apple", "Banana", "Cherry"]; // Array

let person = { name: "John", age: 30, city: "New York" }; // Object

Arrays store **ordered lists**, while objects store **named properties**.

CHAPTER 2: WORKING WITH ARRAYS IN JAVASCRIPT

2.1 Declaring and Accessing Arrays

- **Ways to Declare an Array:**

let numbers = [1, 2, 3, 4, 5]; // Using square brackets (Recommended)

let colors = new Array("Red", "Green", "Blue"); // Using the Array constructor

- **Accessing Array Elements:**

console.log(numbers[0]); // Output: 1

console.log(colors[1]); // Output: Green

Indexes **start from 0**, so the first item is numbers[0].

2.2 Array Methods in JavaScript

Adding and Removing Elements

- **Add elements:**

let animals = ["Dog", "Cat"];

animals.push("Elephant"); // Adds at the end

animals.unshift("Lion"); // Adds at the beginning

console.log(animals); // Output: ["Lion", "Dog", "Cat", "Elephant"]

- **Remove elements:**

animals.pop(); // Removes last element

animals.shift(); // Removes first element

console.log(animals); // Output: ["Dog", "Cat"]

Modifying and Iterating Arrays

- **Modifying an element:**

```
let fruits = ["Apple", "Banana", "Cherry"];
fruits[1] = "Blueberry"; // Changes "Banana" to "Blueberry"
console.log(fruits); // Output: ["Apple", "Blueberry", "Cherry"]
```

- **Looping through an array using forEach():**

```
fruits.forEach(fruit => console.log(fruit));
```

Searching in Arrays

- **Checking if an item exists (includes)**

```
console.log(fruits.includes("Apple")); // Output: true
```

- **Finding an element (find)**

```
let numbers = [10, 20, 30, 40];
let result = numbers.find(num => num > 25);
console.log(result); // Output: 30
```

- **Filtering an array (filter)**

```
let evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers); // Output: [10, 20, 30, 40]
```

CHAPTER 3: WORKING WITH OBJECTS IN JAVASCRIPT

3.1 Declaring and Accessing Objects

- **Creating an Object:**

```
let person = {
  name: "Alice",
  age: 25,
  city: "Paris"
};
```

- **Accessing Object Properties:**

```
console.log(person.name); // Output: Alice
console.log(person["age"]); // Output: 25
```

3.2 Adding, Modifying, and Deleting Object Properties

- **Adding a New Property:**

```
person.country = "France";
console.log(person); // Output: {name: "Alice", age: 25, city: "Paris", country: "France"}
```

- **Modifying an Existing Property:**

```
person.age = 26;
console.log(person.age); // Output: 26
```

- **Deleting a Property:**

```
delete person.city;
console.log(person); // Output: {name: "Alice", age: 26, country: "France"}
```

3.3 Iterating Through an Object

- **Using for...in Loop:**

```
for (let key in person) {
  console.log(key + ": " + person[key]);
}
```

This loops through **all properties** of the object.

- **Using Object.keys() and Object.values()**

```
console.log(Object.keys(person)); // Output: ["name", "age", "country"]
console.log(Object.values(person)); // Output: ["Alice", 26, "France"]
```

CHAPTER 4: CONVERTING BETWEEN ARRAYS AND OBJECTS

4.1 Converting an Object into an Array

- **Extract keys and values from an object:**

```
let entries = Object.entries(person);
console.log(entries);
// Output: [["name", "Alice"], ["age", 26], ["country", "France"]]
```

- **Converting an Array into an Object:**

```
let arr = [["name", "Bob"], ["age", 30]];
let obj = Object.fromEntries(arr);
console.log(obj); // Output: { name: "Bob", age: 30 }
```

Case Study: How E-commerce Websites Use Arrays & Objects

Challenges Faced

- Managing **thousands of product listings**.
- Allowing users to **filter, sort, and search** products.
- Storing user information and shopping carts efficiently.

Solutions Implemented

- Used **arrays** to store product lists.
- Used **objects** to store individual product details.
- Implemented **array methods (filter, map, reduce)** for searching and sorting.

◆ **Example: Filtering Products in an E-commerce Store**

```
let products = [
  { id: 1, name: "Laptop", price: 1000 },
  { id: 2, name: "Phone", price: 500 },
  { id: 3, name: "Tablet", price: 700 }
];

let expensiveProducts = products.filter(p => p.price > 600);
console.log(expensiveProducts);
// Output: [{ id: 1, name: "Laptop", price: 1000 }, { id: 3, name: "Tablet", price: 700 }]
```

📝 Exercise

✅ **Create an array of five car brands** and print each brand using a loop.
✅ **Write a function** that adds a new property (color) to an object.
✅ **Convert an object into an array** using Object.entries().
✅ **Use filter() to find numbers greater than 10** in an array [3, 7, 12, 18, 5].

Conclusion

- **Arrays store lists of items**, while **objects store key-value pairs**.
- **Array methods (push, pop, filter, map)** help manipulate data easily.
- **Objects allow flexible data storage** for user profiles, products, and settings.
- **Converting between arrays and objects** is useful for data manipulation.

LOOPING THROUGH DATA & OBJECT MANIPULATION

CHAPTER 1: UNDERSTANDING LOOPS FOR DATA PROCESSING

1.1 What is Looping in Programming?

Looping is a fundamental concept in programming that allows executing a block of code **multiple times**. It is especially useful when working with **data structures like arrays and objects,** where you need to process multiple items efficiently.

- **Why Use Loops for Data Processing?**
  - Automates **repetitive tasks**.
  - Reduces **manual code duplication**.
  - Improves **performance when handling large datasets**.
- **Example: Looping Through an Array**

```
const numbers = [1, 2, 3, 4, 5];


for (let i = 0; i < numbers.length; i++) {
  console.log(numbers[i]);
}
```

This prints each number **one by one**.

CHAPTER 2: DIFFERENT TYPES OF LOOPS FOR ITERATING THROUGH DATA

2.1 The for Loop

The for loop is best used when the **number of iterations is known**.

- **Example: Looping Through an Array of Names**

```
const names = ["Alice", "Bob", "Charlie"];


for (let i = 0; i < names.length; i++) {
   console.log("Hello, " + names[i]);
}
```

Output:

Hello, Alice

Hello, Bob

Hello, Charlie

2.2 The forEach() Method

The forEach() loop is a **simpler way** to iterate over arrays in JavaScript.

- **Example:**

```
const numbers = [10, 20, 30];


numbers.forEach(function(num) {
```

```
  console.log(num * 2);
});
```

Output:

20

40

60

2.3 The for...of Loop (For Arrays & Iterables)

The for...of loop **directly iterates** over array values.

- **Example:**

```
const colors = ["Red", "Green", "Blue"];

for (let color of colors) {
  console.log(color);
}
```

Output:

Red

Green

Blue

2.4 The while Loop

The while loop is used **when the number of iterations is not known** in advance.

- **Example: Loop Until a Condition is Met**

```
let i = 0;


while (i < 3) {
  console.log("Count: " + i);
  i++;
}
```

Output:

Count: 0

Count: 1

Count: 2

2.5 The do...while Loop

The do...while loop **executes at least once,** even if the condition is false.

- **Example:**

```
let i = 5;


do {
  console.log(i);
```

```
  i++;
} while (i < 3);
```

Output:

5

(The loop runs **once** before checking the condition.)

CHAPTER 3: OBJECT MANIPULATION USING LOOPS

3.1 Looping Through Objects Using for...in

Since objects do not have a **numeric index** like arrays, the for...in loop is used to **iterate through object properties**.

- **Example: Iterating Over an Object**

```
const student = {
  name: "Alice",
  age: 22,
  course: "Computer Science"
};

for (let key in student) {
  console.log(key + ": " + student[key]);
}
```

Output:

name: Alice

age: 22

course: Computer Science

3.2 Converting Objects into Arrays for Easier Looping

JavaScript provides methods to **convert objects into arrays** for easier iteration.

- **Example: Looping Through Object Keys**

```
const car = {
  brand: "Tesla",
  model: "Model 3",
  year: 2023
};

Object.keys(car).forEach(key => {
  console.log(key + ": " + car[key]);
});
```

This extracts object **keys** and loops through them.

- **Example: Looping Through Object Values**

```
Object.values(car).forEach(value => {
  console.log(value);
```

});

This prints only the **values** from the object.

- **Example: Looping Through Both Keys and Values**

```
Object.entries(car).forEach(([key, value]) => {
  console.log(`${key}: ${value}`);
});
```

Output:

brand: Tesla

model: Model 3

year: 2023

CHAPTER 4: COMBINING LOOPS & OBJECT MANIPULATION FOR DATA PROCESSING

4.1 Filtering Data Using Loops

- **Example: Filtering an Array Using forEach()**

```
const numbers = [10, 25, 30, 45];

numbers.forEach(num => {
  if (num > 20) {
    console.log(num);
  }
```

});

Output:

25

30

45

This prints **only numbers greater than 20**.

4.2 Mapping Data for Transformations

The map() function **creates a new array by transforming each element**.

- **Example: Capitalizing Strings in an Array**

```
const lowercaseNames = ["john", "alice", "peter"];


const uppercaseNames = lowercaseNames.map(name =>
name.toUpperCase());


console.log(uppercaseNames);
```

Output:

["JOHN", "ALICE", "PETER"]

4.3 Sorting Data Using sort()

- **Example: Sorting an Array of Numbers**

```
const scores = [85, 42, 96, 58];


scores.sort((a, b) => a - b); // Ascending order

console.log(scores);
```

Output:

```
[42, 58, 85, 96]
```

Case Study: How E-Commerce Websites Use Loops & Object Manipulation

Challenges Faced in E-Commerce

- **Displaying thousands of products dynamically.**
- **Sorting and filtering product listings.**
- **Managing shopping cart data.**

Solutions Implemented

- **Used loops** to iterate through large product catalogs.
- **Applied object manipulation** to filter products based on user preferences.

◆ **Example: Filtering Products Based on Price**

```
const products = [
  { name: "Laptop", price: 1200 },
  { name: "Phone", price: 800 },
  { name: "Tablet", price: 500 }
```

```
];


const affordableProducts = products.filter(product => product.price < 1000);


console.log(affordableProducts);
```

Output:

[{ name: "Phone", price: 800 }, { name: "Tablet", price: 500 }]

This **filters out expensive products**, showing only affordable ones.

- **Key Takeaways from E-Commerce Data Handling:**
  - **Loops efficiently process large datasets.**
  - **Object manipulation helps structure and organize data dynamically.**

📝 Exercise

✅ Write a function that loops through an array of numbers and prints only **even numbers**.
✅ Create an object representing a **student's grades** and loop through it to calculate the **average score**.
✅ Sort an array of **product prices in descending order** using sort().

Conclusion

- **Loops help automate repetitive tasks efficiently.**

- **Object manipulation allows structuring and filtering complex data.**
- **Combining loops and object methods (forEach, map, filter) enhances data processing capabilities.**

SELECTING & MODIFYING ELEMENTS WITH JQUERY

CHAPTER 1: INTRODUCTION TO JQUERY FOR ELEMENT SELECTION AND MODIFICATION

1.1 What is jQuery?

jQuery is a **fast, lightweight, and feature-rich JavaScript library** that simplifies **HTML DOM manipulation, event handling, animations, and AJAX interactions**.

- **Why Use jQuery?**
  - **Easier syntax** compared to vanilla JavaScript.
  - **Cross-browser compatibility**, ensuring code runs consistently.
  - **Powerful selectors** for quickly targeting elements.
- **How to Include jQuery in a Project?**

1️⃣ **CDN Method (Recommended)** → Add this inside <head>:

```
<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
```

2️⃣ **Download Method** → Download jquery.min.js and include:

```
<script src="js/jquery.min.js"></script>
```

- **Basic jQuery Syntax:**

```
$(selector).action();
```

- **$** → The jQuery object.
- **selector** → Targets elements.

- **action()** → Defines the operation to perform.

📌 **Example: Changing the text of an <h1> element using jQuery.**

```
<h1 id="title">Welcome</h1>

<button id="changeText">Change Text</button>


<script>
  $("#changeText").click(function() {
    $("#title").text("Hello, jQuery!");
  });
</script>
```

📌 **Clicking the button updates the <h1> text.**

CHAPTER 2: SELECTING ELEMENTS WITH JQUERY

2.1 Selecting Elements by Tag, Class, and ID

jQuery selectors allow selecting elements **efficiently and dynamically**.

- **Common Selectors:**

Selector	Description	Example
$("p")	Selects all <p> elements.	$("p").hide();
$(".class")	Selects all elements with a class.	$(".box").css("color", "red");

$("#id")	Selects an element by ID.	$("#header").fadeOut();

- **Example: Hiding all paragraphs using jQuery.**

```
$("p").hide();
```

2.2 Selecting Elements Using Attribute and Pseudo-Selectors

- **Advanced jQuery Selectors:**

Selector	Description	Example
[type="text"]	Selects all <input type="text"> elements.	$("[type='text']").val("Hello!");
:first	Selects the first matched element.	$("p:first").css("color", "blue");
:nth-child(n)	Selects the nth child of a parent.	$("li:nth-child(2)").css("font-weight", "bold");

- **Example: Selecting all text inputs and modifying values.**

```
$("input[type='text']").val("Default Text");
```

CHAPTER 3: MODIFYING ELEMENTS WITH JQUERY

3.1 Changing Content Dynamically

- **Methods to Modify Text and HTML Content:**

Method	Description	Example

.text()	Gets or sets text content.	$("#title").text("New Title");
.html()	Gets or sets HTML content.	$("#content").html("Bold Text");
.val()	Gets or sets form input values.	$("#name").val("John Doe");

- **Example: Updating content inside a .**

```
<div id="content">Original Content</div>

<button id="updateBtn">Update Content</button>


<script>
  $("#updateBtn").click(function() {
    $("#content").html("<b>New Content with Bold Text</b>");
  });
</script>
```

📌 **Clicking the button updates the text inside .**

3.2 Changing CSS Styles Dynamically

- **Methods to Modify Styles:**

Method	Description	Example
.css()	Modifies CSS properties.	$("#box").css("color", "red");

.addClass()	Adds a class to an element.	$("p").addClass("highlight");
.removeClass()	Removes a class.	$("#header").removeClass("large-text");
.toggleClass()	Toggles a class.	$("#box").toggleClass("hidden");

- **Example: Changing CSS Styles with jQuery.**

```
<p id="text">This is a paragraph.</p>

<button id="changeColor">Change Color</button>


<script>
  $("#changeColor").click(function() {
    $("#text").css("color", "blue");
  });
</script>
```

📌 **Clicking the button changes the text color to blue.**

3.3 Adding and Removing Elements Dynamically

- **Creating New Elements:**

```
let newItem = $("<li>New Item</li>");
$("#list").append(newItem); // Adds to end of list
```

```
$("#list").prepend(newItem); // Adds to beginning of list
```

- **Removing Elements:**

```
$("#list li:last").remove(); // Removes last list item
```

- **Example: Adding a new list item dynamically.**

```
<ul id="list">
  <li>Item 1</li>
  <li>Item 2</li>
</ul>
<button id="addItem">Add Item</button>

<script>
  $("#addItem").click(function() {
    $("#list").append("<li>New Item</li>");
  });
</script>
```

📌 **Clicking the button appends a new <li> item.**

Case Study: How Amazon Uses jQuery for Real-Time UI Updates

Challenges Faced

- Updating product details **without refreshing the page**.
- Creating **real-time cart modifications** based on user actions.

Solutions Implemented

- Used **jQuery selectors to retrieve product details dynamically**.
- Applied **event listeners for cart updates**.

🔹 **Key Takeaways from Amazon's Success:**

- **jQuery makes UI updates seamless and efficient.**
- **Dynamically selecting and modifying elements improves user experience.**

📝 Exercise

✅ Use jQuery to **hide and show a paragraph** when a button is clicked.
✅ Write jQuery code to **change the text of a <h1> element** dynamically.
✅ Implement a button that **adds and removes new list items dynamically**.

Conclusion

✓ **jQuery provides powerful selectors** to target and modify elements dynamically.
✓ **Using .text(), .html(), and .css() allows dynamic content and style updates.**
✓ **Event listeners enhance user interaction without needing page reloads.**
✓ **Dynamically adding, removing, and modifying elements improves web interactivity.**

EVENT HANDLING & ANIMATIONS WITH JQUERY

CHAPTER 1: INTRODUCTION TO JQUERY EVENT HANDLING & ANIMATIONS

1.1 What is jQuery?

jQuery is a **lightweight JavaScript library** that simplifies event handling, animations, and DOM manipulation.

- **Why Use jQuery for Event Handling & Animations?**
  - Provides **shorter and cleaner syntax** compared to vanilla JavaScript.
  - Supports **cross-browser compatibility**, ensuring consistent performance.
  - Offers **built-in animation methods** for smooth visual effects.
- **Basic jQuery Syntax:**

```
$(document).ready(function() {
  // jQuery code goes here
});
```

This ensures the **DOM is fully loaded** before executing jQuery scripts.

CHAPTER 2: HANDLING EVENTS WITH JQUERY

2.1 Using the .on() Method for Event Handling

The .on() method is the preferred way to handle events in jQuery.

- **Example: Click Event**

```
<button id="clickBtn">Click Me</button>

<p id="message"></p>


<script>

$(document).ready(function() {

  $("#clickBtn").on("click", function() {

    $("#message").text("Button Clicked!");

  });

});

</script>
```

This displays "Button Clicked!" when the button is clicked.

2.2 Handling Mouse Events (mouseenter, mouseleave)

Mouse events enhance **hover effects, tooltips, and dynamic content changes**.

- **Example: Changing Background on Hover**

```
<div id="hoverBox" style="width:200px; height:100px; background:gray;">

  Hover Over Me

</div>


<script>

$(document).ready(function() {
```

```
  $("#hoverBox").mouseenter(function() {
    $(this).css("background", "blue");
  }).mouseleave(function() {
    $(this).css("background", "gray");
  });
});
</script>
```

This changes the background color **when hovering over the box**.

CHAPTER 3: FORM EVENTS & KEYBOARD INTERACTIONS

3.1 Handling Form Submission Events

jQuery can be used to **prevent form submission** and validate inputs.

- **Example: Preventing Form Submission**

```
<form id="myForm">
  <input type="text" id="nameInput" placeholder="Enter your name">
  <button type="submit">Submit</button>
</form>

<script>
$(document).ready(function() {
  $("#myForm").submit(function(event) {
    event.preventDefault();
```

```
    alert("Form submission prevented!");
  });
});
</script>
```

This **prevents the page from refreshing** when the form is submitted.

3.2 Detecting Key Presses (keydown, keyup)

Keyboard events capture **user input in real-time**.

- **Example: Displaying Keystrokes on Screen**

```
<input type="text" id="textInput" placeholder="Type something">
<p id="output"></p>

<script>
$(document).ready(function() {
  $("#textInput").keyup(function() {
    $("#output").text("You typed: " + $(this).val());
  });
});
</script>
```

This **displays the typed text** in real time.

CHAPTER 4: JQUERY ANIMATIONS

4.1 Show & Hide Elements with Animations

jQuery provides **built-in methods** to create animations like fadeIn(), fadeOut(), slideToggle(), etc.

- **Example: Toggling Visibility with a Button Click**

```
<button id="toggleBtn">Toggle Paragraph</button>

<p id="togglePara">This paragraph will toggle.</p>


<script>

$(document).ready(function() {

  $("#toggleBtn").click(function() {

    $("#togglePara").slideToggle();

  });

});

</script>
```

This makes the paragraph **appear/disappear** when the button is clicked.

4.2 Animating Elements (animate())

The .animate() function allows **custom animations**.

- **Example: Moving a Box**

```
<button id="moveBtn">Move Box</button>

<div id="box" style="width:50px; height:50px; background:red; position:absolute;"></div>


<script>
```

```
$(document).ready(function() {
  $("#moveBtn").click(function() {
    $("#box").animate({left: "250px"}, 1000);
  });
});
</script>
```

This moves the box **250px to the right** in **1 second**.

Case Study: How Facebook Uses jQuery for Interactive UI

Challenges Faced

- Ensuring **smooth UI interactions**.
- Reducing **manual clicks** by making actions intuitive.

Solutions Implemented

- Used **jQuery click events** for toggling comments and likes.
- Implemented **fade and slide effects** for pop-ups and notifications.

🔹 **Key Takeaways from Facebook's UI Strategy:**

- **Smooth interactions improve user engagement.**
- **Real-time event handling enhances the experience.**

📝 **Exercise**

✅ Create a **click event** to toggle a sidebar using .slideToggle().
✅ Implement an **input field** that detects keystrokes and displays

them on the page.

✅ Design an **animated button** that grows in size when hovered over.

Conclusion

- **jQuery simplifies event handling and animations** with shorter syntax.
- **Methods like .on(), .animate(), and .fadeIn() improve UI interactions.**
- **Using event delegation ensures efficient and scalable scripts.**

FETCHING DATA FROM AN API

CHAPTER 1: INTRODUCTION TO APIS AND DATA FETCHING

1.1 What is an API?

An **API (Application Programming Interface)** is a way for applications to **communicate and exchange data** over the internet. APIs allow developers to fetch data from external servers, databases, or web services.

- **Why Use APIs?**
  - Fetch **real-time data** (e.g., weather updates, stock prices, news).
  - Connect to **third-party services** like Google Maps, OpenWeather, or Twitter.
  - Build **dynamic web applications** that update content without reloading.
- **Types of APIs:**

1. **REST API** – Uses HTTP methods like GET, POST, PUT, DELETE.
2. **GraphQL API** – Allows fetching specific data instead of entire datasets.
3. **SOAP API** – A legacy protocol using XML-based requests.

CHAPTER 2: FETCHING DATA USING THE FETCH API

2.1 What is the Fetch API?

The **Fetch API** is a modern JavaScript feature that allows making **asynchronous requests** to retrieve data from a web server.

- **Basic Syntax of Fetch API:**

```
fetch("https://api.example.com/data")
  .then(response => response.json()) // Convert response to JSON
  .then(data => console.log(data)) // Log the fetched data
  .catch(error => console.log("Error:", error)); // Handle errors
```

2.2 Making a Simple GET Request

- **Example: Fetching Random Users from an API**

```
fetch("https://randomuser.me/api/")
  .then(response => response.json())
  .then(data => console.log(data.results[0]))
  .catch(error => console.error("Error:", error));
```

This retrieves a **random user's details** from an API and logs them to the console.

2.3 Handling API Responses Properly

Every API request returns a **response object**, which contains:

- status → HTTP status code (200 OK, 404 Not Found, 500 Internal Server Error).
- json() → Converts API response to JavaScript object.

- **Example: Checking API Response Status**

```
fetch("https://api.example.com/data")
  .then(response => {
```

```
    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.log("Error fetching data:", error));
```

This ensures that if the **API request fails**, the error is handled gracefully.

CHAPTER 3: FETCHING DATA USING ASYNC/AWAIT

3.1 Why Use Async/Await?

- **Advantages of Async/Await:**
  - Makes code **look cleaner and easier to read**.
  - Avoids deeply nested .then() chains (callback hell).
- **Example: Fetching Data with Async/Await**

```
async function fetchUser() {
  try {
    let response = await fetch("https://randomuser.me/api/");
    let data = await response.json();
    console.log(data.results[0]); // Logs the user data
  } catch (error) {
    console.error("Error fetching data:", error);
```

```
  }
}
```

```
fetchUser();
```

Using **async/await**, we **wait for the response** before proceeding to the next step.

3.2 Fetching Multiple API Endpoints

- **Example: Fetching User Data and Weather Data Simultaneously**

```
async function fetchMultipleAPIs() {
  try {
    let userResponse = await fetch("https://randomuser.me/api/");
    let weatherResponse = await
fetch("https://api.weatherapi.com/v1/current.json?key=YOUR_KEY
&q=London");

    let userData = await userResponse.json();
    let weatherData = await weatherResponse.json();

    console.log("User:", userData.results[0]);
    console.log("Weather:", weatherData);
  } catch (error) {
    console.error("Error:", error);
```

```
  }
}
```

```
fetchMultipleAPIs();
```

This **fetches user details and weather data simultaneously**.

CHAPTER 4: POSTING DATA TO AN API

4.1 Sending Data Using a POST Request

A POST request is used when **sending data to an API**, such as submitting a form.

- **Example: Sending Form Data to an API**

```
async function submitData() {
  let user = { name: "Alice", age: 25 };

  try {
    let response = await fetch("https://example.com/api/users", {
      method: "POST",
      headers: {
        "Content-Type": "application/json"
      },
      body: JSON.stringify(user)
    });
```

```
    let data = await response.json();
    console.log("User Created:", data);
  } catch (error) {
    console.error("Error:", error);
  }
}

submitData();
```

This **submits user details** to an API using a POST request.

4.2 Handling API Errors and Timeouts

Sometimes, an API **takes too long to respond**, or the **server is down**.

- **Example: Setting a Timeout for API Requests**

```
async function fetchWithTimeout(url, timeout = 5000) {
  const controller = new AbortController();
  const signal = controller.signal;

  setTimeout(() => controller.abort(), timeout);

  try {
    let response = await fetch(url, { signal });
    let data = await response.json();
```

```
    console.log(data);
  } catch (error) {
    console.error("Request timeout or error:", error);
  }
}

fetchWithTimeout("https://api.example.com/data");
```

This **cancels a request if it takes more than 5 seconds**.

Case Study: How E-Commerce Websites Use API Fetching

Challenges Faced

- Displaying **real-time product listings** from a database.
- Fetching **customer reviews and ratings dynamically**.
- Handling **large amounts of data efficiently**.

Solutions Implemented

- Used **Fetch API to retrieve product data dynamically**.
- Implemented **search functionality using API filtering**.
- Applied **caching mechanisms to reduce API calls**.

◆ **Example: Fetching Product Data for an E-commerce Website**

```
async function fetchProducts() {
  try {
    let response = await fetch("https://fakestoreapi.com/products");
```

```
    let products = await response.json();
    console.log(products);
  } catch (error) {
    console.error("Error fetching products:", error);
  }
}

fetchProducts();
```

This retrieves **a list of products** from a **fake store API**.

📝 Exercise

✅ **Make a GET request** to fetch a list of users from https://jsonplaceholder.typicode.com/users.
✅ **Write a function** that fetches weather data from OpenWeather API.
✅ **Create a form** that sends user data via a POST request using fetch().
✅ **Handle API errors** and display an error message in the UI if fetching fails.

Conclusion

- **APIs allow applications to communicate and exchange data dynamically.**
- **Fetch API provides a simple way to retrieve data using GET and POST requests.**

- **Async/Await improves readability when handling API calls.**
- **Error handling and timeouts ensure a smooth user experience.**

DISPLAYING DYNAMIC CONTENT ON WEB PAGES

CHAPTER 1: INTRODUCTION TO DYNAMIC CONTENT

1.1 What is Dynamic Content?

Dynamic content refers to **web content that updates or changes without requiring a full page reload**. Unlike static content, which remains the same unless manually updated, dynamic content is **fetched, generated, or modified using JavaScript, APIs, or databases**.

🔹 **Why is Dynamic Content Important?**

- Improves **user experience** by updating content in real-time.
- Enables **personalization** (e.g., greeting users by name).
- Fetches **live data** (e.g., news feeds, stock prices, weather updates).

🔹 **Example: Static vs. Dynamic Content**

📌 **Static Content (Manual Updates Required)**

```
<p>Welcome, John!</p>
```

📌 **Dynamic Content (Updated Automatically)**

```
<p id="greeting"></p>


<script>

  document.getElementById("greeting").innerText = "Welcome, John!";
```

</script>

The second example **dynamically updates** the content using JavaScript.

CHAPTER 2: METHODS FOR DISPLAYING DYNAMIC CONTENT

2.1 Using JavaScript to Update Content

JavaScript allows **modifying web content dynamically** using the **Document Object Model (DOM)**.

- **Example: Updating an HTML Element's Content**

```
<p id="message">Old Message</p>


<script>
  document.getElementById("message").innerText = "New Message Updated!";
</script>
```

- **Example: Adding New Content Dynamically**

```
<div id="container"></div>


<script>
  let newElement = document.createElement("p");
  newElement.innerText = "This content was added dynamically!";

document.getElementById("container").appendChild(newElement);
</script>
```

This script **creates and inserts** a new <p> tag into the webpage.

2.2 Using JavaScript Loops to Display Dynamic Lists

Loops allow **displaying multiple elements dynamically** (e.g., listing products, news articles).

- **Example: Displaying a List of Items Dynamically**

```
<ul id="itemList"></ul>


<script>
  const items = ["Apple", "Banana", "Cherry"];
  let listContainer = document.getElementById("itemList");


  items.forEach(item => {
    let li = document.createElement("li");
    li.innerText = item;
    listContainer.appendChild(li);
  });
</script>
```

This **loops through an array** and displays each item inside a <ul> list dynamically.

CHAPTER 3: FETCHING AND DISPLAYING DATA FROM AN API

3.1 What is an API?

An **API (Application Programming Interface)** allows web pages to **fetch and display real-time data** from external sources (e.g., weather apps, stock prices).

- **Why Use APIs for Dynamic Content?**
  - Retrieves **real-time data** without manually updating content.
  - Enables **live updates** (e.g., sports scores, stock markets).
  - Integrates **third-party services** like Google Maps, OpenWeather, etc.

3.2 Fetching Data from an API and Displaying it on a Web Page

JavaScript's fetch() function retrieves **external data** and injects it into an HTML page.

- **Example: Fetching and Displaying User Data from an API**

```
<div id="userContainer"></div>


<script>
  fetch("https://jsonplaceholder.typicode.com/users/1")
  .then(response => response.json())
  .then(data => {
    document.getElementById("userContainer").innerHTML = `
      <h3>${data.name}</h3>
      <p>Email: ${data.email}</p>
      <p>City: ${data.address.city}</p>
    `;
```

```
});
</script>
```

This script **fetches user details from an API** and displays them inside a div.

Chapter 4: Handling User Input for Dynamic Content

4.1 Updating Content Based on User Input

Forms and buttons allow **users to update content dynamically**.

- **Example: Updating a Heading Based on User Input**

```
<input type="text" id="nameInput" placeholder="Enter your name">
<button onclick="updateHeading()">Submit</button>
<h2 id="userHeading">Hello, User!</h2>

<script>
  function updateHeading() {
    let name = document.getElementById("nameInput").value;
    document.getElementById("userHeading").innerText = "Hello, " + name + "!";
  }
</script>
```

Users enter their name, and the heading updates **instantly**.

4.2 Showing/ Hiding Elements Dynamically

- **Example: Displaying a Message on Button Click**

```
<button onclick="showMessage()">Click Me</button>

<p id="message" style="display: none;">You clicked the button!</p>


<script>
  function showMessage() {
    document.getElementById("message").style.display = "block";
  }
</script>
```

This script **reveals a hidden paragraph** when the button is clicked.

CHAPTER 5: REAL-WORLD APPLICATIONS OF DYNAMIC CONTENT

5.1 Dynamic Product Listings in E-Commerce

E-commerce websites dynamically **display products based on user preferences and stock availability**.

- **Example: Displaying Products from an Array**

```
<div id="productList"></div>


<script>
  const products = [
    { name: "Laptop", price: "$999" },
    { name: "Smartphone", price: "$699" }
  ];
```

```
let productContainer = document.getElementById("productList");


products.forEach(product => {
    let div = document.createElement("div");
    div.innerHTML = `<h4>${product.name}</h4><p>Price: ${product.price}</p>`;
    productContainer.appendChild(div);
});
</script>
```

This script **automatically generates product listings** from an array.

5.2 Live Data Updates (Stock Market, Weather, News)

Dynamic content is crucial for **live feeds** like:

- **Stock Prices**
- **Weather Forecasts**
- **Breaking News**

◆ **Example: Fetching Live Weather Data**

```
<div id="weather"></div>


<script>
```

```
fetch("https://api.openweathermap.org/data/2.5/weather?q=London&appid=YOUR_API_KEY")
  .then(response => response.json())
  .then(data => {
    document.getElementById("weather").innerHTML = `
      <h3>Weather in ${data.name}</h3>
      <p>Temperature: ${(data.main.temp - 273.15).toFixed(1)}°C</p>
      <p>Condition: ${data.weather[0].description}</p>
    `;
  });
</script>
```

This fetches **live weather data from an API** and displays it dynamically.

Case Study: How Facebook Uses Dynamic Content for Real-Time Updates

Challenges Faced by Facebook

- Keeping **news feeds updated without page reloads**.
- Handling **millions of real-time interactions** (likes, comments, messages).

Solutions Implemented

- **JavaScript and AJAX** fetch new posts dynamically.

- **WebSockets** enable live updates for messages and notifications.

🔹 **Key Takeaways from Facebook's Dynamic Content Strategy:**

- **Efficient dynamic updates improve engagement and UX.**
- **Using APIs and JavaScript ensures real-time interactivity.**

📝 **Exercise**

✅ Use **JavaScript** to dynamically update the content of a <p> tag every 5 seconds.
✅ Fetch and display **a random user's name and email** from an API.
✅ Build a **search feature** that filters and displays results dynamically.

Conclusion

- **Dynamic content enhances user experience by updating data in real time.**
- **JavaScript and APIs allow seamless content updates without page reloads.**
- **Fetching data from APIs enables live updates for weather, stocks, and news.**
- **Interactivity improves engagement by responding to user input dynamically.**

ASSIGNMENT:

BUILD A DYNAMIC WEBPAGE USING JAVASCRIPT, JQUERY, AND API INTEGRATION

Step-by-Step Guide to Building a Dynamic Webpage Using JavaScript, jQuery, and API Integration

---

📌 Step 1: Setting Up the Project

1.1 Create Project Files

1. **Create a project folder** (e.g., dynamic-webpage).
2. **Inside the folder, create the following files:**
   - index.html ⟶ Main HTML file.
   - style.css ⟶ CSS file for styling.
   - script.js ⟶ JavaScript file for interactivity and API integration.

📌 Step 2: Creating the HTML Structure

🛠 Define the Basic Structure in index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Dynamic Webpage</title>
```

```
  <link rel="stylesheet" href="style.css">
  <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
<!-- jQuery CDN -->
</head>
<body>

  <div class="container">
    <h1>Weather Information</h1>
    <input type="text" id="cityInput" placeholder="Enter City
Name">
    <button id="getWeather">Get Weather</button>

    <div id="weatherData"></div>
  </div>

  <script src="script.js"></script>
</body>
</html>
```

📌 **This creates a simple input field and a button for fetching weather data dynamically.**

📌 Step 3: Styling the Webpage

🛠 Define Styles in style.css

```
body {
    font-family: Arial, sans-serif;
    background-color: #f4f4f4;
    text-align: center;
}

.container {
    width: 50%;
    margin: 50px auto;
    padding: 20px;
    background: white;
    box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.1);
    border-radius: 5px;
}

h1 {
    color: #333;
}

input {
    width: 80%;
    padding: 10px;
```

```
    margin: 10px 0;
    border: 1px solid #ccc;
    border-radius: 5px;
}

button {
    background-color: #28a745;
    color: white;
    border: none;
    padding: 10px 15px;
    cursor: pointer;
    border-radius: 5px;
}

button:hover {
    background-color: #218838;
}

#weatherData {
    margin-top: 20px;
    padding: 10px;
    background-color: #e3e3e3;
```

```
  border-radius: 5px;
}
```

📌 **This improves the UI for a clean and modern look.**

📌 Step 4: Implementing JavaScript and jQuery for Interactivity

🛠 Add Script Logic in script.js

```
$(document).ready(function () {
  $("#getWeather").click(function () {
    let city = $("#cityInput").val();

    if (city === "") {
      alert("Please enter a city name");
      return;
    }

    let apiKey = "your_api_key_here";  // Replace with a real API key
    let url = `https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${apiKey}&units=metric`;

    // Fetch API Data using jQuery AJAX
    $.ajax({
      url: url,
```

```
        type: "GET",
        success: function (response) {
          let weatherInfo = `
            <h2>${response.name}, ${response.sys.country}</h2>
            <p><strong>Temperature:</strong> ${response.main.temp}°C</p>
            <p><strong>Weather:</strong> ${response.weather[0].description}</p>
            <p><strong>Humidity:</strong> ${response.main.humidity}%</p>
          `;
          $("#weatherData").html(weatherInfo);
        },
        error: function () {
          $("#weatherData").html("<p>City not found. Please try again.</p>");
        }
      });
  });
});
```

📌 **How This Works:**

✓ Uses **jQuery AJAX** to fetch weather data from the OpenWeather API.

✓ Displays **city, temperature, weather description, and humidity**.

✓ Handles **errors** if the city is not found.

📌 Step 5: Testing & Debugging

✅ Steps to Test the Webpage

1. **Enter a city name** (e.g., "London") and click the button.
2. **Check if weather data appears.**
3. **Enter an incorrect city name** (e.g., "abcd") and verify error handling.

✅ Expected Behaviors

✓ If a **valid city** is entered, **weather data is displayed dynamically**.
✓ If an **invalid city** is entered, an **error message appears**.

📌 Step 6: Enhancing the Webpage with More Features

1️⃣ Add a Loading Indicator

Modify script.js to show a **loading message** while fetching data.

```
$("#getWeather").click(function () {
    $("#weatherData").html("<p>Loading...</p>");
});
```

2️⃣ Save Last Searched City in Local Storage

Modify script.js to **store the last searched city** and **load it on page reload**.

```
$(document).ready(function () {
    let savedCity = localStorage.getItem("lastCity");
    if (savedCity) {
```

```
    $("#cityInput").val(savedCity);
  }

  $("#getWeather").click(function () {
    let city = $("#cityInput").val();
    localStorage.setItem("lastCity", city);
  });
});
```

📌 **This saves the last searched city and auto-fills it on the next visit.**

🎯 Final Outcome

✓ **A fully interactive webpage** using JavaScript and jQuery.
✓ **Live API integration** fetching real-time weather data.
✓ **Error handling for incorrect city names.**
✓ **Local Storage implementation to save last searched city.**

📌 Bonus: Expanding the Project

1. Allow users to get their current location's weather.

```
navigator.geolocation.getCurrentPosition(function(position) {
  let lat = position.coords.latitude;
  let lon = position.coords.longitude;
});
```

2. Add a dropdown for selecting favorite cities.

```
<select id="cityDropdown">
  <option value="New York">New York</option>
  <option value="Paris">Paris</option>
  <option value="Tokyo">Tokyo</option>
</select>
```

3. Improve UI with Bootstrap or Tailwind CSS for better styling.

Conclusion

✓ **Built a dynamic webpage** with JavaScript and jQuery.
✓ **Integrated an external API** for real-time weather data.
✓ **Implemented Local Storage** to remember the last searched city.
✓ **Handled errors and improved UX** with loading indicators and auto-fill.