

## Agenda

- jQuery
- AJAX
- Promise

## jQuery

- jQuery is a fast, small, and feature-rich JavaScript library.
- It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers.
- It is a lightweight, "write less, do more", JavaScript library.
- With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript.
- The jQuery library contains the following features:
  1. HTML/DOM manipulation
  2. CSS manipulation
  3. HTML event methods
  4. Effects and animations
  5. AJAX

## Adding jQuery

- The jQuery library is a single JavaScript file, and you reference it with the HTML `<script>` tag
- make sure that the script tag should be inside the head section ``HTML

...

## Basic Syntax

- `$(selector).action()`
  - \$ sign to define/access jQuery
  - (selector) to "query (or find)" HTML elements
  - action() to be performed on the element(s)
- Example
  - `$(this).hide()` - hides the current element.
  - `$("p").hide()` - hides all elements.
  - `$(".testclass").hide()` - hides all elements with class="testclass".
  - `$("#testId").hide()` - hides the element with id="testId".

## AJAX

- AJAX stands for Asynchronous JavaScript and XML.
- It is a technique used in web development to update parts of a web page without reloading the entire page.
- By using AJAX, web applications can send and retrieve data asynchronously from a server in the background, improving the user experience by making the app feel faster and more interactive.
- The basic flow of an AJAX request:

#### 1. User Interaction:

- A user performs an action (e.g., clicking a button).

#### 2. JavaScript Sends a Request:

- JavaScript sends an HTTP request to the server using the XMLHttpRequest object or the newer fetch API.

#### 3. Server Processes the Request:

- The server processes the request (e.g., fetches data from a database) and sends back a response.

#### 4. JavaScript Updates the Web Page:

- JavaScript processes the server's response and updates the web page dynamically, without requiring a page reload.

## JSON

- JSON (JavaScript Object Notation) is a lightweight data format used for storing and exchanging data between a server and a web application.
- It is easy to read and write for humans and machines.
- JSON data is written in key-value pairs and uses {} for objects and [] for arrays.
- the data in the form of JSON is converted entirely in the string (using JSON.stringify()) and then sent through the request or response.
- once we receive the request on server or the response on client we convert the string data back into JSON format (using JSON.parse())

```
{
  "name": "John Doe",
  "age": 30,
  "email": "john@example.com",
  "isStudent": false
}

[
  {
    "id": 1,
    "name": "Alice"
  },
  {
    "id": 2,
    "name": "Bob"
  }
]
```

```
}  
]
```

## XHR (XMLHttpRequest)

- It is a JavaScript API that allows developers to make HTTP requests (to fetch data, send data, etc.) from a web server without reloading the page.
- It is commonly used for building AJAX functionality.

## Steps to Use XHR

### 1. Create an XHR Object:

- Use `let helper = new XMLHttpRequest();`

### 2. Configure the Request:

- Specify the request method (GET, POST, etc.) and the URL
- `helper.open(method, url, async)`.
- `async: Boolean, true (asynchronous, default) or false (synchronous).`

### 3. Send the Request:

- `helper.send()` to send the request.

### 4. Handle the Response:

- `helper.onreadystatechange` or `helper.onload` to process the server's response.

## using jQuery Ajax method

## Promise

- A Promise in JavaScript is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value.
- Promises were introduced to solve the lot of callback problem, which occurs when multiple asynchronous operations are nested, leading to messy, hard-to-read, and error-prone code.
- A Promise has three possible states:
  1. Pending: The initial state when the Promise is neither fulfilled nor rejected.
  2. Fulfilled: The operation completed successfully, and the Promise has a resolved value.
  3. Rejected: The operation failed, and the Promise has a reason (error).
- Once a Promise is either fulfilled or rejected, it becomes settled, and its state will not change again.

## Reasons for using Promises:

1. Cleaner Code and Readability: Promises provide a structured way to handle asynchronous code, leading to more readable and maintainable code compared to nested callbacks (often referred to as "callback hell" or "pyramid of doom").

```
// Callback-based (less readable)
getData(function(a) {
  processData(a, function(b) {
    displayResult(b, function(c) {
      console.log(c);
    });
  });
});

// Promise-based (more readable and chainable)
getData()
  .then(processData)
  .then(displayResult)
  .then(console.log)
  .catch(handleError);
```

2. Improved Error Handling: Promises offer a centralized and robust mechanism for error handling using the `.catch()` method. This simplifies error management across a chain of asynchronous operations, unlike callbacks where error handling can become cumbersome and scattered.
3. Better Control over Asynchronous Flow: Promises represent a value that might not be available immediately, allowing you to attach handlers for both successful completion (fulfillment) and failure (rejection) of an asynchronous action. This provides more explicit control over the flow of asynchronous operations.
4. Enabling Async/Await: Promises are the fundamental building block for `async/await` syntax, which further simplifies asynchronous code by allowing it to be written in a more synchronous-looking style, improving readability and reducing cognitive load.

```
async function fetchDataAndProcess() {
  try {
    const data = await getData();
    const processedData = await processData(data);
    const result = await displayResult(processedData);
    console.log(result);
  } catch (error) {
    console.error("An error occurred:", error);
  }
}
```

## Fetch API

- It is a modern alternative to XHR.
- Fetch uses Promises, making the code cleaner and easier to manage.
- Easier to read and manage with `.then()` and `.catch()` or `async/await`.
- No need for `xhr.open()`, `xhr.send()`, or `xhr.onload`.
- All configurations are included in the `fetch()` function.
- Use `.catch()` to handle errors, such as network issues or HTTP errors.

- Built-in JSON Support: Use `json()` to parse JSON responses easily.

```
//using then() and catch()
function btn1Clicked() {
  fetch("http://127.0.0.1:5500/data.json", { "method": "GET" })
  .then((response) => {
    if (response.ok)
      return response.json()
    else
      throw new Error("Something went wrong")
  })
  .then((data) => {
    console.log(data)
  })
  .catch((error) => {
    console.log("error = " + error)
  })
}

//using async/await
async function btn2Clicked() {
  try {
    let result = await fetch("http://127.0.0.1:5500/data.json", { "method": "GET" })
  }
  if (result.ok) {
    const data = await result.json()
    console.log(data)
  }
  else
    throw new Error("Something went wrong")
} catch (error) {
  console.log("error -" + error)
}
}
```

- Both `async/await` and `.then()` are ways to handle Promises in JavaScript, but they differ in syntax, readability, and how they handle asynchronous code.
- Which One to Use?

#### 1. Use Async/Await:

- When you have multiple asynchronous tasks that need to be executed in sequence.
- When you want better readability and maintainability.
- When you want to handle errors more cleanly with `try/catch`.

#### 2. Use `.then()`:

- When you're working with simpler tasks (single or few Promises).
- When you need backward compatibility (older browsers without `async/await`).