Week 11: JS APIs

Amr A Khllaf

June 21, 2025

Contents

1 Rev on AddEventListener & What is the ContextMenu & PreventDefault?

1.1 AddEventListener Method

- i need when i click on any place on the page, it will log the event

```
document.addEventListener("click", function (event) {
  console.log("Hello", event);
```

```
});
```

1.2 ContextMenu Event

- The context menu is the menu that appears when you right-click on an element in a web page.

```
document.addEventListener("contextmenu", function (event) {
  console.log("Context menu opened", event);
});
```

1.3 PreventDefault Method (Event Method)

- The `preventDefault` method (**Event Method**) is used to prevent the default action of an event from occurring.
- It's commonly used with events like `click`, `submit`, and `contextmenu` to stop the browser from performing its default behavior.

```
document.addEventListener("contextmenu", function (event) {
  event.preventDefault();
  console.log("Context menu opened", event);
});
```

1.4 Example of PreventDefault with a Form Submission

```
document.addEventListener("submit", function (event) {
  event.preventDefault();
  console.log("Form submission canceled", event);
});
```

1.5 Example of PreventDefault with a Click Event

```
document.addEventListener("click", function (event) {
  event.preventDefault();
  console.log("Click event canceled", event);
});
```

---

2 API Introduction

2.1 What is an API?

- API Uses JSON (JavaScript Object Notation) to exchange data between the client and server, It's the Communication Language between 2 Different Applications.
- Request and Response are the 2 Main Parts of an API to deal with Backend.
- An API (**Application Programming Interface**) is a set of rules and protocols that allows different software applications to communicate with each other.

- APIs define the methods and data formats that applications can use to request and exchange information.
- APIs can be used to access web services, databases, and other resources.
- They can be public (open to anyone) or private (restricted to specific users or applications).

2.2 Types of APIs

- **Web APIs**: Allow communication between web servers and clients (e.g., RESTful APIs, GraphQL).
- **Library APIs**: Provide functions and methods for developers to use in their applications (e.g., jQuery, React).
- **Operating System APIs**: Allow applications to interact with the underlying operating system (e.g., Windows API, POSIX).
- **Hardware APIs**: Enable communication with hardware devices (e.g., USB, Bluetooth).

2.3 How APIs Work

- APIs work by defining a set of endpoints (URLs) that applications can use to send requests and receive responses.
- Each endpoint corresponds to a specific function or resource within the API.
- APIs typically use standard HTTP methods (GET, POST, PUT, DELETE) to perform actions on the resources.
- Responses are usually in **JSON** or **XML** format, making it easy for applications to parse and use the data.

---

3 API is link and What is consist of & How We Get Api From Backend?

3.1 API Link Structure

- An API link (or endpoint) typically consists of the following components:
  - **Base URL**: The root URL of the API (e.g., `https://api.example.com`).
    - It's the **Domain Name** of the API.
    - Domain is the server address where the API is hosted.
    - **Version**: Sometimes included in the URL to specify the version of the API (e.g., `/v1`).
    - Example: `https://api.example.com/v1`
  - **Path (endpoint)**: The specific resource or action being accessed (e.g., `/users`, `/posts/1`).
  - **Query Parameters**: Optional parameters that can be included in the URL to filter or modify the request (e.g., `?sort=asc&limit=10`).
- **Example API Link**: `https://api.example.com/users?sort=asc&limit=10`

3.2 API Methods

- APIs use different HTTP methods to perform actions on resources. The most common methods are:
  - **GET**: Retrieve data from the server (e.g., get a list of users).
  - **POST**: Send data to the server to create a new resource (e.g., create a new user).
  - **PUT**: Update an existing resource on the server (e.g., update user information).
  - **DELETE**: Remove a resource from the server (e.g., delete a user).

3.3 How to Get an API from the Backend

- To get an API from the backend, you typically need to:
  1. **Define the API Requirements**: Determine what data or functionality you need from the backend.
  2. **Create Endpoints**: Design the endpoints that will provide the required data or perform actions.
  3. **Implement the Logic**: Write the server-side code to handle requests and responses for each endpoint.
  4. **Test the API**: Use tools like Postman or cURL to test the API endpoints and ensure they work as expected.
  5. **Document the API**: Provide clear documentation for developers on how to use the API, including endpoints, request methods, parameters, and response formats.

3.4 JSON Format

- JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy to read and write for humans and machines.
- It is commonly used in APIs to exchange data between the client and server.
- JSON consists of key-value pairs and can represent complex data structures like arrays and objects.
- **Example JSON**:

```
{
  "users": [
    {
      "id": 1,
      "name": "John Doe",
      "email": "john.doe@example.com"
    }
  ]
}
```

3.5 JSON Structure

- JSON is structured as a collection of key-value pairs, where keys are strings and values can be strings, numbers, arrays, objects, or booleans.
- It is often used to represent data in a hierarchical format, making it easy to parse and manipulate.

- JSON is language-independent, meaning it can be used with various programming languages, including JavaScript, Python, Java, and more.

---

4 Testing APIs with Postman

4.1 What is Postman?

- Postman is a popular API development and testing tool that allows developers to send requests to APIs, view responses, and debug issues.
- It provides a user-friendly interface for creating, testing, and documenting APIs without writing code.
- Postman supports various HTTP methods (GET, POST, PUT, DELETE) and allows you to add headers, query parameters, and request bodies.
- It also provides features like environment variables, collections, and automated testing to streamline the API development process.

4.2 How to Use Postman

1. **Install Postman**: Download and install Postman from the official website.
2. **Create a New Request**: Open Postman and create a new request by selecting the HTTP method (GET, POST, etc.) and entering the API endpoint URL.
3. **Add Headers and Parameters**: If required, add headers (e.g., `Content-Type`, `Authorization`) and query parameters to the request.
4. **Send the Request**: Click the "Send" button to send the request to the API.
5. **View the Response**: Postman will display the response from the API, including the status code, headers, and body.
6. **Test the API**: You can write tests in Postman to validate the response data, check status codes, and ensure the API behaves as expected.
7. **Save Requests**: Save your requests in collections for future use and organization.
8. **Document the API**: Use Postman's documentation features to create clear and comprehensive API documentation for your team or users.

4.3 Example of Using Postman

- **GET Request**: To retrieve a list of users from an API, you would set the method to GET and enter the endpoint URL (e.g., `https://api.example.com/users`).
- **POST Request**: To create a new user, you would set the method to POST, enter the endpoint URL (e.g., `https://api.example.com/users`), and provide the user data in the request body in JSON format.

```
{
  "name": "Jane Doe",
  "email": "jane.doe@example.com"
}
```

- **Response**: After sending the request, Postman will display the response from the API, including the status code (e.g., 201 Created) and the created user data.

```
{
  "id": 2,
  "name": "Jane Doe",
  "email": "jane.doe@example.com"
}
```

4.4 Postman Interface

- Params:
  - Used to add query parameters to the request URL.
  - Example: `?sort=asc&limit=10`
- Authorization:
  - Used to set authentication credentials for the API request.
  - Common methods include Basic Auth, Bearer Token, and OAuth.
- Headers (Meta Information):
  - Used to add custom headers to the request.
  - Example: `Content-Type: application/json`
- Body:
  - Used to send data in the request body, typically for POST and PUT requests.
  - Can be in various formats, including JSON, form-data, or x-www-form-urlencoded.
  - raw: Used to send raw data in the request body, such as JSON or XML.
- Pre-request Script:
  - Used to write JavaScript code that runs before the request is sent.
  - Useful for setting dynamic variables or modifying the request.
- Tests:
  - Used to write tests to validate the API response.
  - You can use JavaScript to check status codes, response data, and more.
- Settings:
  - Used to configure Postman settings, such as request timeout, proxy settings, and environment variables.

4.5 Types of Responses

- **200 OK**: The request was successful, and the server returned the requested data.
- **201 Created**: The request was successful, and a new resource was created (commonly used with POST requests).
- **204 No Content**: The request was successful, but there is no content to return (commonly used with DELETE requests).
- **400 Bad Request**: The server could not understand the request due to invalid syntax.
- **401 Unauthorized**: The request requires authentication, and the user is not authenticated or does not have permission.

- **403 Forbidden**: The server understood the request, but the user does not have permission to access the resource.
- **404 Not Found**: The requested resource could not be found on the server.
- **500 Internal Server Error**: The server encountered an unexpected condition that prevented it from fulfilling the request.
- **502 Bad Gateway**: The server received an invalid response from an upstream server while trying to fulfill the request.
- **503 Service Unavailable**: The server is currently unable to handle the request due to temporary overload or maintenance.
- **504 Gateway Timeout**: The server did not receive a timely response from an upstream server while trying to fulfill the request.
- **429 Too Many Requests**: The user has sent too many requests in a given amount of time, often used for rate limiting.

4.6 Response Shape

- The response shape refers to the structure of the data returned by an API in response to a request.
- It typically includes the following components:
  - **Status Code**: Indicates the result of the request (e.g., 200, 404, 500).
  - **Headers**: Metadata about the response, such as content type and caching information.
  - **Body**: The actual data returned by the API, often in JSON or XML format.
- The response shape can vary depending on the API and the specific endpoint being accessed.

4.7 Example Response Shape

```
{
  "status": "success",
  "data": {
    "users": [
      {
        "id": 1,
        "name": "John Doe",
        "email": "john.doe@example.com"
      },

      {
        "id": 2,
        "name": "Jane Doe",
        "email": "jane.doe@example.com"
      }
    ]
  }
}
```

- In this example, the response shape includes a status field indicating success, and a data field containing an array of user objects with their respective properties (id, name, email).

---

5 Json Placeholder & How to use it in JS?

5.1 What is JSON Placeholder Site?

- JSON Placeholder is a free online REST API that provides fake data for testing and prototyping purposes.
- It allows developers to simulate API requests and responses without needing a real backend server.
- JSON Placeholder provides endpoints for common resources like users, posts, comments, and more.
- It is commonly used for learning, testing, and prototyping applications that require API interactions.

5.2 How to Use JSON Placeholder in JavaScript

- There is Builtin Object in JS Called **XMLHttpRequest** that allows you to make HTTP requests to APIs, including JSON Placeholder.
- We don't deal with XMLHttpRequest Directly, We take a Copy of it and use the Fetch API.
- **readyState**: is a property of the XMLHttpRequest object that indicates the current state of the request.
- The readyState property can have the following values:
  - **0 (UNSENT)**: The request has been created but not yet sent.
  - **1 (OPENED)**: The request has been opened but not yet sent.
  - **2 (HEADERS_RECEIVED)**: The request has been sent, and the response headers have been received.
  - **3 (LOADING)**: The response body is being received.
  - **4 (DONE)**: The request is complete, and the response is fully received.

```
let req = new XMLHttpRequest(); // Create a new XMLHttpRequest object
req.open("GET", "https://jsonplaceholder.typicode.com/posts"); // Send a
↪  GET request to the specified URL
req.send(); // Send the request
req.addEventListener("readystatechange", () => {
  if (req.readyState == 4) {
    console.log(JSON.parse(req.response)); // Parse the JSON response
    console.log(typeof req.response); // Array Of Objects
  }
});
```

---

6 ReadyState Values & Status

6.1 ReadyState Values

- The `readyState` property of the XMLHttpRequest object indicates the current state of the request.
- The possible values for `readyState` are:
  - **0 (UNSENT Or Uninitialized)**: The request has been created but not yet sent.
  - **1 (OPENED Or Initialized)**: The request has been opened but not yet sent.
  - **2 (HEADERS_RECEIVED)**: The request has been sent, and the response headers have been received.
  - **3 (LOADING)**: The response body is being received.
  - **4 (DONE)**: The request is complete, and the response is fully received.

6.2 Status Codes

- The `status` property of the XMLHttpRequest object indicates the HTTP status code returned by the server in response to the request.
- Common status codes include:
  - **200 OK**: The request was successful, and the server returned the requested data.
  - **201 Created**: The request was successful, and a new resource was created (commonly used with POST requests).
  - **204 No Content**: The request was successful, but there is no content to return (commonly used with DELETE requests).
  - **400 Bad Request**: The server could not understand the request due to **invalid syntax**.
  - **401 Unauthorized**: The request requires authentication, and the user is not authenticated or does not have permission.
  - **403 Forbidden**: The server understood the request, but the user does not have permission to access the resource.
  - **404 Not Found**: The requested resource could not be found on the server.
  - **500 Internal Server Error**: The server encountered an unexpected condition that prevented it from fulfilling the request.
  - **502 Bad Gateway**: The server received an invalid response from an upstream server while trying to fulfill the request.
  - **503 Service Unavailable**: The server is currently unable to handle the request due to temporary overload or maintenance.
  - **504 Gateway Timeout**: The server did not receive a timely response from an upstream server while trying to fulfill the request.
- **429 Too Many Requests**: The user has sent too many requests in a given amount of time, often used for rate limiting.

```
let req = new XMLHttpRequest(); // Create a new XMLHttpRequest object
req.open("GET", "https://jsonplaceholder.typicode.com/posts"); // Send a
↪  GET request to the specified URL
req.send(); // Send the request
req.addEventListener("readystatechange", () => {
  if (req.readyState == 4 && req.status == 200) {
    console.log(JSON.parse(req.response)); // Parse the JSON response
    console.log(typeof req.response); // Array Of Objects
  } else if (req.readyState == 4) {
    console.error("Error:", req.status, req.statusText); // Log error if
    ↪  status is not 200
  }
});
```

---

7 Note About extension

- You need to install the **JSON Formatter** extension in your browser to view the JSON response in a more readable format.
- This extension formats JSON responses, making them easier to read and understand.
- It highlights syntax, collapses large objects, and provides a tree view for nested structures.
- After installing the extension, you can view JSON responses directly in your browser without needing to parse them manually.

---

8 Display all Posts Example

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
    ↪  />
    <title>APIs Week</title>
    <link rel="stylesheet" href="./css/all.min.css" />
    <link rel="stylesheet" href="./css/bootstrap.min.css" />
    <link rel="stylesheet" href="./css/style.css" />
  </head>
  <body>
    <div class=" allPosts w-50 m-auto vh-100 bg-secondary
    ↪  overflow-auto"></div>
    <script src="./js/all.min.js"></script>
    <script src="./js/bootstrap.bundle.min.js"></script>
    <script src="./js/main.js"></script>
  </body>
</html>
```

```
var allPosts = [];
```

```
let req = new XMLHttpRequest(); // Create a new XMLHttpRequest object
req.open("GET", "https://jsonplaceholder.typicode.com/posts"); // Send a
↪  GET request to the specified URL
req.send(); // Send the request
req.addEventListener("readystatechange", () => {
  if (req.readyState == 4 && req.status == 200) {
    console.log(JSON.parse(req.response)); // Parse the JSON response
    console.log(typeof req.response); // Array Of Objects
    allPosts = JSON.parse(req.response);
    displayAllPosts();
  } else if (req.readyState == 4) {
    console.error("Error:", req.status, req.statusText); // Log error if
    ↪  status is not 200
  }
});

function displayAllPosts() {
  let cartona = "";
  for (let i = 0; i < allPosts.length; i++) {
    cartona += `<div class="post bg-white rounded-2 m-3 p-3 ">
                <h4>${allPosts[i].title}</h4>
                <p>${allPosts[i].body}</p>
            </div>`;
  }
  document.querySelector(".allPosts").innerHTML = cartona;
}
```

---

9 News API Site and What is Live Server and CORS ?

- We have a New API Free Site Called **News API** News API Site
- It provides access to news articles from various sources and allows you to filter news by categories, sources, and more.
- You can use the News API to fetch the latest news articles, search for specific topics, and display them in your application.

9.1 Live Server

- Live Server is a development tool that allows you to run a local web server and view your web pages in real-time.
- It automatically refreshes the browser whenever you make changes to your HTML, CSS, or JavaScript files.
- Live Server is commonly used in web development to speed up the development process by providing instant feedback on changes.
- You can install Live Server as a Visual Studio Code extension or use other similar tools like Browsersync or Webpack Dev Server.

9.2 CORS (Cross-Origin Resource Sharing)

- CORS is a security feature implemented by web browsers to prevent unauthorized access to resources from different origins (domains).
- It allows servers to specify which origins are allowed to access their resources.
- CORS is important when making API requests from a web application to a different domain (cross-origin requests).
- If the server does not allow cross-origin requests, the browser will block the request and throw a CORS error.
    - **Error 426**: CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.
- To resolve CORS issues, the server must include the appropriate headers in its response to allow cross-origin requests.
- Common CORS headers include:
    - **Access-Control-Allow-Origin**: Specifies which origins are allowed to access the resource (e.g., `*` for all origins).
    - **Access-Control-Allow-Methods**: Specifies which HTTP methods are allowed (e.g., `GET`, `POST`, `PUT`, `DELETE`).
    - **Access-Control-Allow-Headers**: Specifies which headers can be included in the request.

9.3 Example on News API

- Html File

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
    ↪  />
    <title>APIs Week</title>
    <link rel="stylesheet" href="./css/all.min.css" />
    <link rel="stylesheet" href="./css/bootstrap.min.css" />
    <link rel="stylesheet" href="./css/style.css" />
  </head>
  <body>
    <div class="container">
      <div class="row news"></div>
    </div>
    <script src="./js/all.min.js"></script>
    <script src="./js/bootstrap.bundle.min.js"></script>
    <script src="./js/main.js"></script>
  </body>
</html>
```

JavaScript File

```
let allNews = [];
let newsReq = new XMLHttpRequest();
newsReq.open(
  "GET",

  ↪  "https://newsapi.org/v2/top-headlines?country=us&category=business&apiKey=cd678
);
newsReq.send();
newsReq.addEventListener("readystatechange", () => {
  if (newsReq.readyState == 4 && newsReq.status == 200) {
    console.log(JSON.parse(newsReq.response).articles);
    allNews = JSON.parse(newsReq.response).articles;
    displayUsNews();
  } else if (newsReq.readyState == 4) {
    console.error("Error:", newsReq.status, newsReq.statusText); // Log
    ↪  error if status is not 200
  }
});

function displayUsNews() {
  let cartona = "";

  for (let i = 0; i < allNews.length; i++) {
    cartona += ` <div class="col-md-3">
                    <a href="${
                      allNews[i].url
                    }" target="_blank" class="text-decoration-none">
                        <img src="${
                          allNews[i].urlToImage == null
                            ? "./images/worldimage.jpeg"
                            : allNews[i].urlToImage
                        }" class="w-100" alt="Us News">
                    </a>
                    <h3>${
                      allNews[i].title == null ? "No Title" :
↪  allNews[i].title
                    }</h3>
                    <p>${
                      allNews[i].description == null
                        ? "No Description"
                        : allNews[i].description
                    }</p>
                </div>`;
  }
  document.querySelector(".row.news").innerHTML = cartona;
}
```

---

10 Hint About Params (Query Params) Values & Exercise on News API

- HTML File

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
    ↪  />
    <title>APIs Week</title>
    <link rel="stylesheet" href="./css/all.min.css" />
    <link rel="stylesheet" href="./css/bootstrap.min.css" />
    <link rel="stylesheet" href="./css/style.css" />
  </head>
  <body>
    <div class="container overflow-auto">
      <input
        onkeydown="getNews(this.value)"
        type="text"
        class="form-control w-75 m-auto mt-3 "
        placeholder="Search for posts..."
      />
      <!--! this.value: is the input value -->
      <div class="row news my-4"></div>
    </div>
    <div class=" allPosts w-50 m-auto vh-100 bg-secondary
    ↪  overflow-auto"></div>
    <script src="./js/all.min.js"></script>
    <script src="./js/bootstrap.bundle.min.js"></script>
    <script src="./js/main.js"></script>
  </body>
</html>
```

- JavaScript File

```
let allNews = [];

function getNews(term) {
  let newsReq = new XMLHttpRequest();
  newsReq.open(
    "GET",

    ↪  `https://newsapi.org/v2/everything?q=${term}&from=2025-06-17&to=2025-06-17&so
  );
  newsReq.send();
  console.log("newReq Sent");

  newsReq.addEventListener("readystatechange", () => {
    if (newsReq.readyState == 4 && newsReq.status == 200) {
```

```
      console.log(JSON.parse(newsReq.response).articles);
      allNews = JSON.parse(newsReq.response).articles;

      displayUsNews();
    } else if (newsReq.readyState == 4) {
      console.log("Hi From Error");

      console.error("Error:", newsReq.status, newsReq.statusText); // Log
      ↪  error if status is not 200
    }
  });
}

function displayUsNews() {
  let cartona = "";

  for (let i = 0; i < allNews.length; i++) {
    cartona += ` <div class="col-md-3">
                    <a href="${
                      allNews[i].url
                    }" target="_blank" class="text-decoration-none">
                        <img src="${
                          allNews[i].urlToImage == null
                            ? "./images/worldimage.jpeg"
                            : allNews[i].urlToImage
                        }" class="w-100" alt="Us News">
                    </a>
                    <h3>${
                      allNews[i].title == null ? "No Title" :
↪   allNews[i].title
                    }</h3>
                    <p>${
                      allNews[i].description == null
                        ? "No Description"
                        : allNews[i].description
                    }</p>
                </div>`;
  }
  document.querySelector(".row.news").innerHTML = cartona;
}
```

- In the API Link there is a Query Params Called `q` that is used to search for posts.
- The `q` parameter is used to specify the search term for the news articles.
- You can change the value of `q` to search for different topics or keywords.

---

11 Forkify API Site

Forkify API Site

- Forkify is a free online API that provides access to a collection of recipes and allows you to search for recipes by ingredients, cuisine, and more.
- It is commonly used for building recipe applications and websites.
- You can use the Forkify API to fetch recipe data, search for specific recipes, and display them in your application.
- The Forkify API provides endpoints for searching recipes, getting recipe details, and more.
- It is a great resource for learning how to work with APIs and build applications that consume recipe data.

12 Loadend Event Vs. ReadyStateChange Event

- The `loadend` event is triggered when the request has completed, regardless of whether it was successful or not.
- It indicates that the request has finished, and you can access the response data.
- The `readystatechange` event is triggered when the readyState of the XMLHttpRequest object changes, indicating the progress of the request.

12.1 Why We Always Check on the ReadyStateChange Event == 4?

- We check for `readyState == 4` to ensure that the request has completed and the response is fully received before processing the data.
- This ensures that we only attempt to access the response data when it is available and ready to be used.
- Because the code will execute 3 times before reaching the final state, we need to ensure we are only working with the completed response.

12.2 Example of Using `loadend` Event

```
let req = new XMLHttpRequest();
req.open("GET", "https://jsonplaceholder.typicode.com/posts");
req.send();
req.addEventListener("loadend", () => {
  if (req.status == 200) {
    console.log("Success:", JSON.parse(req.response));
  } else {
    console.error("Error:", req.status, req.statusText);
  }
});
```

12.3 Example of Using `readystatechange` Event

```
let req = new XMLHttpRequest();
req.open("GET", "https://jsonplaceholder.typicode.com/posts");
req.send();
req.addEventListener("readystatechange", () => {
  if (req.readyState == 4 && req.status == 200) {
    console.log("Success:", JSON.parse(req.response));
```

```
  } else if (req.readyState == 4) {
    console.error("Error:", req.status, req.statusText);
  }
});
```

13 Note About Conditional Statements Decision Making

- Use the `if` statement, along with `else if` and `else`, when you need to handle multiple exclusive conditions and make a crucial decision in your code.
- If your logic does not require branching into several distinct paths, avoid unnecessary use of `else if` or `else` to keep your code clear and concise.

---

14 Sync vs Async

- **Synchronous (Sync)**: code is executed sequentially, meaning each operation must complete before the next one begins. This can lead to blocking behavior, where the program waits for a long-running task to finish before continuing.
- **Asynchronous (Async)**: code allows operations to run in the background, enabling the program to continue executing without waiting for the task to complete. This is particularly useful for tasks like network requests, file I/O, and timers.
- In JavaScript, asynchronous operations are often handled using callbacks, promises, or async/await syntax.
- Using asynchronous code can improve the responsiveness of your application, especially in scenarios where tasks may take a long time to complete, such as fetching data from an API or performing heavy computations.
- By using asynchronous programming techniques, you can avoid blocking the main thread and keep your application responsive to user interactions.
- In the context of APIs, asynchronous requests allow you to fetch data from a server without freezing the user interface, providing a smoother user experience.
- When working with APIs, it's common to use asynchronous requests to fetch data, process it, and update the UI without blocking the main thread.
- Asynchronous programming is a fundamental concept in modern web development, enabling developers to build efficient and responsive applications that can handle multiple tasks concurrently.

```
console.log("Start"); // Synchronous code starts executing
getNews(); // Asynchronous function call
console.log("End"); // Synchronous code continues executing without
↪  waiting for getNews to finish
```

---

15 Callbacks Concept

- A callback is a function that is passed as an argument to another function and is executed after the completion of that function.

- Callbacks are commonly used in asynchronous programming to handle the result of an operation once it is complete.
- They allow you to define what should happen after a task is finished, such as processing data or updating the UI.
- Callbacks can be used to handle events, such as user interactions or API responses, and are a fundamental concept in JavaScript programming.
- In the context of APIs, callbacks are often used to handle the response from an API request, allowing you to process the data once it is received.
- Callbacks can be defined as anonymous functions or named functions, and they can be passed as arguments to other functions.
- Using callbacks can help you manage asynchronous operations and ensure that your code executes in the correct order, especially when dealing with tasks that may take time to complete, such as network requests or file I/O.
- Callbacks can lead to "callback hell" if not managed properly, where nested callbacks make the code difficult to read and maintain. To avoid this, you can use techniques like Promises or async/await for better readability and error handling.

15.1 Simple Example to Understand Callbacks

```
function one(hamada) {
  console.log("One");
  hamada(); // Call the callback function passed as an argument
  console.log(hamada()); // Call the callback function and log its return
  ↪  value
  console.log("End of One");
}

function two() {
  console.log("two");
}

one(two); // Pass the 'two' function as a callback to 'one'
```

15.2 Example on the callback Concept with API

```
function getPizza(callback) {
  // Fetch pizza recipes from the API, then call the callback when done
  let allPizzas = [];
  let recipeReq = new XMLHttpRequest();

  recipeReq.open("GET",
↪  "https://forkify-api.herokuapp.com/api/search?q=pizza");
  recipeReq.send();

  recipeReq.addEventListener("readystatechange", () => {
    if (recipeReq.readyState == 4 && recipeReq.status) {
      console.log("Hi From Pizzas ");
      console.log(JSON.parse(recipeReq.response).recipes); // This is the
      ↪  Array of Objects
```

```
      allPizzas = JSON.parse(recipeReq.response).recipes;
      callback(); // Call the next function in the chain
    }
  });
}

function getBeef(callback) {
  // Fetch beef recipes from the API, then call the callback when done
  let allBeefs = [];
  let recipeReq = new XMLHttpRequest();

  recipeReq.open("GET",
↪  "https://forkify-api.herokuapp.com/api/search?q=pizza");
  recipeReq.send();

  recipeReq.addEventListener("readystatechange", () => {
    if (recipeReq.readyState == 4 && recipeReq.status) {
      console.log("Hi From Beefs ");
      console.log(JSON.parse(recipeReq.response).recipes); // This is the
      ↪  Array of Objects
      allBeefs = JSON.parse(recipeReq.response).recipes;
      callback(); // Call the next function in the chain
    }
  });
}

function getSalads(callback) {
  // Fetch salad recipes from the API, then call the callback when done
  let allSalads = [];
  let recipeReq = new XMLHttpRequest();

  recipeReq.open("GET",
↪  "https://forkify-api.herokuapp.com/api/search?q=pizza");
  recipeReq.send();
      console.log(JSON.parse(recipeReq.response).recipes); // This is the
      ↪  Array of Objects
      allSalads = JSON.parse(recipeReq.response).recipes;
      callback(); // Call the next function in the chain
    }
  });
}

function endMenu() {
  console.log("Menu Has Ended");
}


// Call Back Hell
getBeef(function () {
  getPizza(function () {
```

```
    getSalads(function () {
      endMenu();
    });
  });
});
```

---

16 Promises Concept (ES6)

- Promise Solve the Problem of Callback Hell.
- A Promise is an Built-in object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value.
- Promises provide a way to handle asynchronous operations in a more manageable and readable way compared to callbacks.
- A Promise can be in one of three states:
  - **Pending**: The initial state, neither fulfilled nor rejected.
  - **Fulfilled**: The operation completed successfully, and the Promise has a resulting value.
  - **Rejected**: The operation failed, and the Promise has a reason for the failure (an error).
- Promises allow you to chain multiple asynchronous operations together using the `.then()` method, which is called when the Promise is fulfilled, and the `.catch()` method, which is called when the Promise is rejected.
- Promises help avoid "callback hell" by providing a cleaner syntax for handling asynchronous operations and making it easier to read and maintain code.
- Promises can be created using the `Promise` constructor, which takes a function (executor) that receives two arguments: `resolve` and `reject`. You call `resolve` when the operation is successful and `reject` when it fails.
- Promises can also be used with the `async/await` syntax, which allows you to write asynchronous code that looks synchronous, making it easier to read and understand.

16.1 Example of Using Promises

```
var x = new Promise(hamada){
  console.log("sayHello");
  hamada();
// This function will be executed every time the browser is refreshed.

x.then(function(){
  console.log("sayGoodBye");
})
}

- Output:
  - sayHello
  - sayGoodBye
```

16.2 Using Our Example on Promises

- Promises allow us to handle asynchronous operations more easily.

```
// Fetch pizza recipes from the API, then call the callback when done
function getPizza() {
  return new Promise(function (callback) {
    // Create a new Promise that will resolve when the data is fetched
    let allPizzas = [];
    let recipeReq = new XMLHttpRequest();

    recipeReq.open(
      "GET",
      "https://forkify-api.herokuapp.com/api/search?q=pizza"
    );
    recipeReq.send();

    recipeReq.addEventListener("readystatechange", () => {
      if (recipeReq.readyState == 4 && recipeReq.status) {
        console.log("Hi From Pizzas ");
        console.log(JSON.parse(recipeReq.response).recipes); // This is the
        ↪  Array of Objects
        allPizzas = JSON.parse(recipeReq.response).recipes;
        callback(); // Call the next function in the chain
      }
    });
  });
}

// Fetch beef recipes from the API, then call the callback when done
function getBeef() {
  return new Promise(function (callback) {
    let allBeefs = [];
    let recipeReq = new XMLHttpRequest();

    recipeReq.open(
      "GET",
      "https://forkify-api.herokuapp.com/api/search?q=beef"
    );
    recipeReq.send();

    recipeReq.addEventListener("readystatechange", () => {
      if (recipeReq.readyState == 4 && recipeReq.status) {
        console.log("Hi From Beefs ");
        console.log(JSON.parse(recipeReq.response).recipes); // This is the
        ↪  Array of Objects
        allBeefs = JSON.parse(recipeReq.response).recipes;
        callback(); // Call the next function in the chain
      }
    });
  });
```

```
}

// Fetch salad recipes from the API, then call the callback when done
function getSalads() {
  return new Promise(function (callback) {
    let allSalads = [];
    let recipeReq = new XMLHttpRequest();

    recipeReq.open(
      "GET",
      "https://forkify-api.herokuapp.com/api/search?q=salad"
    );
    recipeReq.send();

    recipeReq.addEventListener("readystatechange", () => {
      if (recipeReq.readyState == 4 && recipeReq.status) {
        console.log("Hi From Salad");
        console.log(JSON.parse(recipeReq.response).recipes); // This is the
        ↪  Array of Objects
        allSalads = JSON.parse(recipeReq.response).recipes;
        callback(); // Call the next function in the chain
      }
    });
  });
}

function endMenu() {
  console.log("Menu Has Ended");
}

getBeef().then(getPizza).then(getSalads).then(endMenu); // Chaining
↪  Promises To Solve the Callback Hell
```

---

17 Async / Await

- Async/await is a modern way to handle asynchronous operations in JavaScript, making it easier to write and read asynchronous code.
- The `async` keyword is used to define an asynchronous function, which allows you to use the `await` keyword inside it.
- The `await` keyword is used to wait for a Promise to resolve or reject before moving on to the next line of code.

```
await getBeef();
await getPizza();
await getSalads();
endMenu();

// This is will cause a problem : `Uncaught SyntaxError: await is only
↪  valid in async functions and the top level bodies of modules`
```

- To avoid this error you need to wrap the code in an async function:

```
async function fetchData() {
  await getBeef();
  await getPizza();
  await getSalads();
  endMenu();
}
```

- You can then use the self-invoking function to call it immediately:

```
(async function () {
  await getBeef();
  await getPizza();
  await getSalads();
  endMenu();
})();
```

---

18 Fetch Function & What is the init optional parameter?

- The Fetch API is a modern way to make HTTP requests in JavaScript, providing a more powerful and flexible alternative to XMLHttpRequest.
- It allows you to make network requests to servers and handle responses in a more readable and concise way.
- The Fetch API returns a Promise that resolves to the Response object representing the response to the request.
- The Response object contains methods to read the response body, such as `.json()`, `.text()`, and `.blob()`.
- The Fetch API supports various HTTP methods (GET, POST, PUT, DELETE, etc.) and allows you to set headers, query parameters, and request bodies.
- The Fetch API is built into modern browsers and can be used in both client-side and server-side JavaScript environments.

```
(async function () {
  var x = await
  ↪  fetch("https://forkify-api.herokuapp.com/api/search?q=beef");
  var res = await x.json(); // This will return a Promise
  // json() ==> it's a method of the Response object that reads the
  ↪  response body and returns a Promise that resolves to the parsed JSON
  ↪  data.
  console.log(res); // This will log the parsed JSON data
})(); // Self-invoking function to call the async function immediately
```

18.1 Using Options(Init Or Settings) With Fetch

- The Fetch API allows you to specify the HTTP method you want to use for the request by passing an optional `init` object as the second argument to the `fetch()` function.
- The `init` object can include properties such as `method`, `headers`, `body`, and more.

- The `method` property specifies the HTTP method to use (e.g., "GET", "POST", "PUT", "DELETE").
- The `headers` property allows you to set custom headers for the request.

```
(async function () {
  var x = await
  ↪  fetch("https://forkify-api.herokuapp.com/api/search?q=beef", {
    method: "GET",
    headers: {
      "Content-Type": "application/json",
    },
  });
  var res = await x.json(); // This will return a Promise
  console.log(res); // This will log the parsed JSON data
  console.log(x.status); // This will log the HTTP status code of the
  ↪  response
})();
```

---

19 setTimeout & setInterval & How to clear them

- `setTimeout` and `setInterval` are built-in JavaScript functions that allow you to execute code after a specified delay or at regular intervals.
- `setTimeout` is used to execute a function once after a specified delay **(in milliseconds)**.
- `setInterval` is used to repeatedly execute a function at specified intervals **(in milliseconds)**.
- Both functions return a unique identifier (ID) that can be used to cancel the scheduled execution using `clearTimeout` or `clearInterval`.
- setTimeout Example:
  - The Params of the setTimeout Function are:
    - The first parameter is the function to be executed after the delay(function or reference).
    - The second parameter is the delay in milliseconds.

```
setTimeout(function () {
  console.log("Hello after 2 seconds");
}, 2000); // Executes the function after 2000 milliseconds (2 seconds)
```

- setInterval Example:
  - The Params of the setInterval Function are:
    - The first parameter is the function to be executed at regular intervals(function or reference).
    - The second parameter is the interval in milliseconds.

```
setInterval(function () {
  console.log("Hello every 2 seconds");
}, 2000); // Executes the function every 2000 milliseconds (2 seconds)
```

- To cancel a scheduled execution, you can use `clearTimeout` for `setTimeout` and `clearInterval` for `setInterval`.

```
let timeoutId = setTimeout(function () {
  console.log("This will not execute");
}, 2000);
clearTimeout(timeoutId); // Cancels the scheduled execution of the
↪  setTimeout function

let intervalId = setInterval(function () {
  console.log("This will not execute");
}, 2000);
clearInterval(intervalId); // Cancels the scheduled execution of the
↪  setInterval function
```

- Example:

```
let intervalId = setInterval(function () {
  console.log("Welcome");
}, 500);

setTimeout(function(){
  clearInterval(intervalId); // This will stop the interval after 1 second
}, 1000);

---
if(sessionStorage.getItem("test") == null){
  setTimeout(function(){
    sessionStorage.setItem("test", "Hello World");
    console.log("Welcome");
  }, 1000);
}
// Output: Welcome after 1 second

// Explain: The setTimeout function is used to delay the execution of the
↪  code inside it by 1000 milliseconds (1 second). After 1 second, the
↪  sessionStorage is set with the key "test" and the value "Hello World",
↪  and "Welcome" is logged to the console.
// When We refresh the Page the code will not execute again because the
↪  sessionStorage already has the key "test" set to "Hello World".
```

---

20 How the browser deals with our Code

20.1 Call Stack (Execution Stack):

- The call stack is a data structure that keeps track of the function calls in your code.
- When a function is called, it is added to the top of the stack, and when it returns, it is removed from the stack.
- The call stack ensures that functions are executed in the correct order.
- The browser parses the JavaScript code and builds a call stack.

- When it encounters an asynchronous operation (like a fetch request), it offloads that operation to the Web APIs (like the Fetch API).
- The Web APIs handle the request and, once completed, push the callback (or promise resolution) onto the task queue.
- The main thread continues executing the rest of the code without waiting for the async operation to complete.
- When the call stack is empty, the event loop checks the task queue and processes the next task (like the promise resolution).
- This allows the browser to remain responsive and handle multiple tasks concurrently, improving the user experience.

20.2 Web APIs (Has All Async Methods)

- The Web APIs provide a set of functions and objects that allow developers to perform asynchronous operations, such as making network requests, handling timers, and interacting with the DOM.
- These APIs are built into the browser and are accessible from JavaScript code running in the browser environment.
- When an asynchronous operation is initiated (like a fetch request), it is sent to the Web APIs.
- The Web APIs handle the operation in the background and notify the event loop when it is complete.

20.3 Task Queue (Message Queue)

- The task queue is a queue that holds tasks (callbacks or promise resolutions) that are ready to be executed.
- When an asynchronous operation completes, the Web APIs push the callback or promise resolution onto the task queue.
- The event loop continuously checks the call stack and the task queue.
- When the call stack is empty, the event loop processes the next task in the task queue.
- This allows the browser to execute asynchronous code without blocking the main thread, ensuring a smooth user experience.

20.4 Event Loop

- The event loop is a mechanism that continuously checks the call stack and the task queue.
- It ensures that the browser can handle asynchronous operations without blocking the main thread.
- The event loop checks if the call stack is empty and, if so, processes the next task in the task queue.
- The event loop allows the browser to remain responsive and handle multiple tasks concurrently, improving the user experience.

20.4.1 Interview Question

20.5 What Executes First: Synchronous or Asynchronous Code?

- **Synchronous (Sync) code** is always executed first. This means all statements and function calls that do not involve asynchronous operations will run in order, one after another, blocking further execution until each completes.
- **Asynchronous (Async) code**—such as API requests, timers, or event handlers—runs in the background. The browser offloads these tasks to Web APIs, and their callbacks or promise resolutions are queued for execution only after the synchronous code and the current call stack are finished.
- In summary:
  - The synchronous code runs to completion first.
  - Asynchronous code executes later, when the call stack is empty and the event loop picks up tasks from the queue.
- This ensures that your main program logic is handled before any async results are processed.

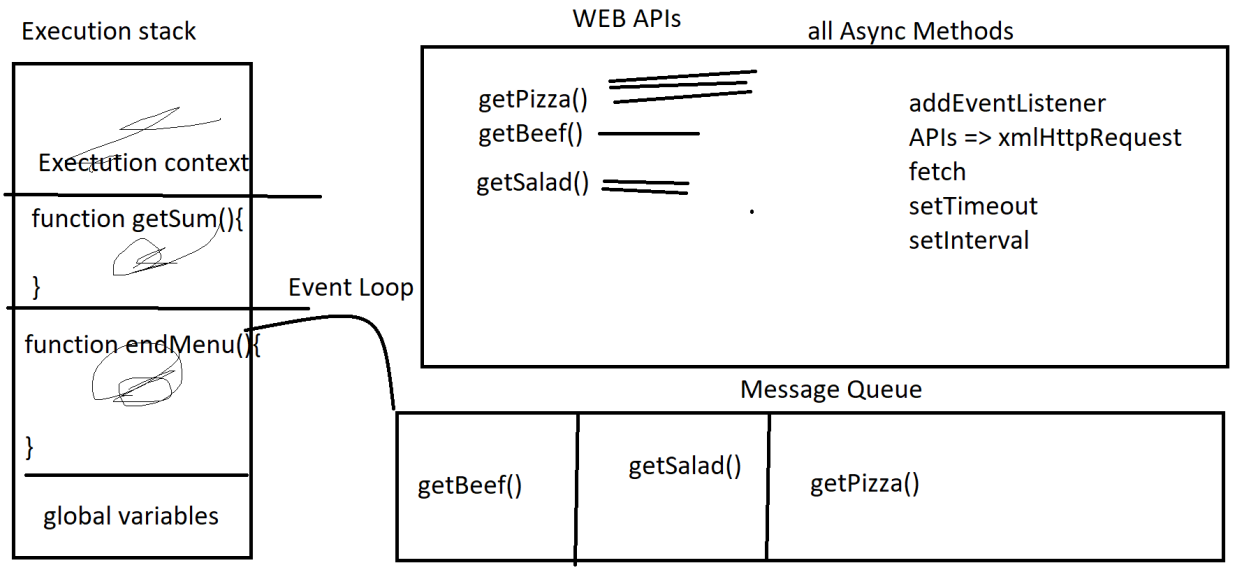

Figure 1: How Browser Deal With The Code
