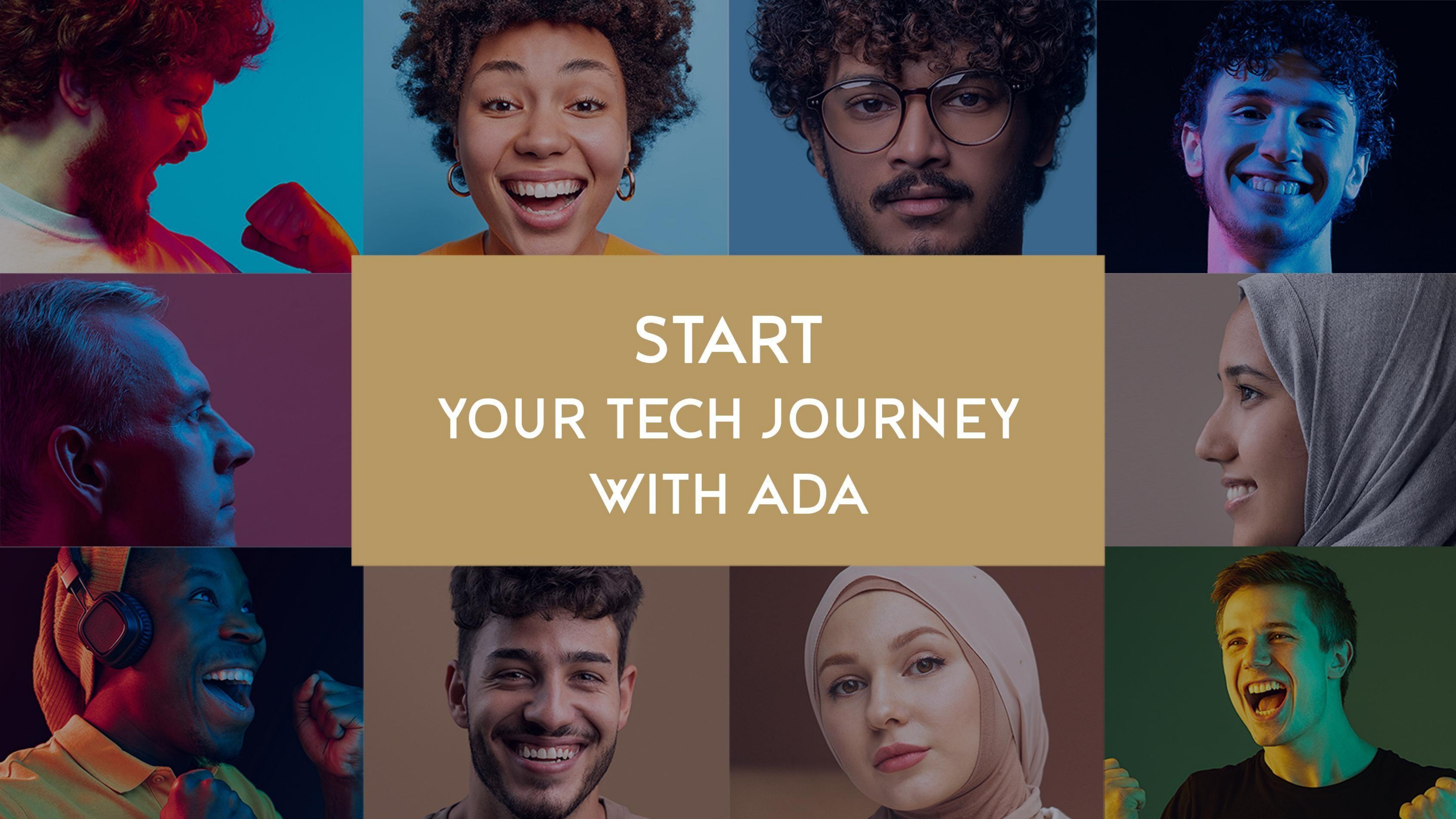
ACADEMY OF DIGITAL ARTS EGYPT

Google Cisco Adobe IBM Microsoft CompTIA



START YOUR TECH JOURNEY WITH ADA

Agenda

- Events (quick overview) - Event-driven programming basics
- Callbacks (introduction only) - Understanding asynchronous execution
- Promises - Modern approach to handling async operations
- Async & Await - Clean, readable asynchronous code
- Fetch API / Axios - Making HTTP requests (GET, POST)
- Working with real APIs and handling responses
- Hands-on exercise: Use JSONPlaceholder API to fetch and display posts

Introduction to Asynchronous Programming

The Problem with Synchronous Code:

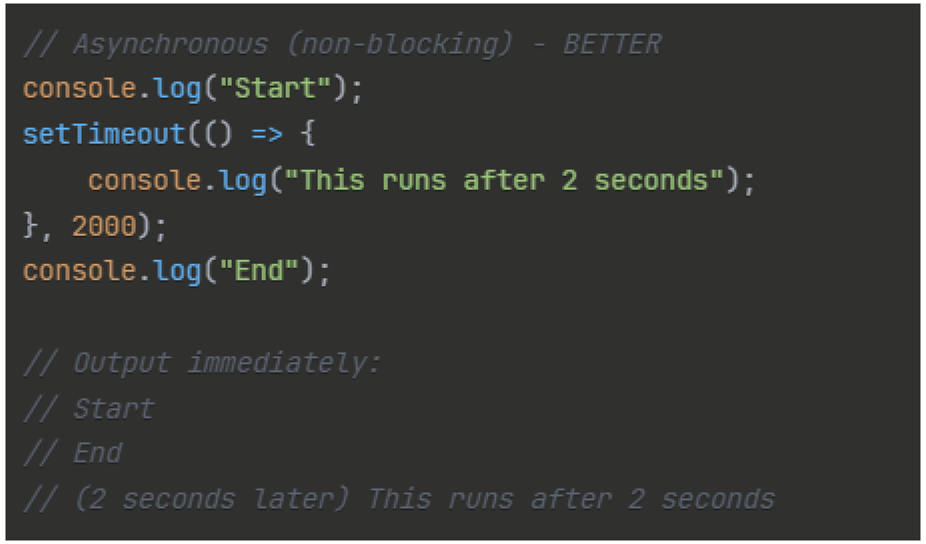
```
// Synchronous (blocking) - PROBLEMATIC
console.log("Start");
waitFor5Seconds(); // This blocks everything!
console.log("End");

// User can't interact with the app for 5 seconds!
```

Real-World Scenarios That Take Time:

- File operations - Reading/writing files
- Network requests - API calls, downloading data
- Database operations - Querying, inserting data
- User input - Waiting for clicks, form submissions
- Timers - setTimeout, setInterval

Introduction to Asynchronous Programming

The Solution - Asynchronous Programming:



```
// Asynchronous (non-blocking) - BETTER
console.log("Start");
setTimeout(() => {
    console.log("This runs after 2 seconds");
}, 2000);
console.log("End");

// Output immediately:
// Start
// End
// (2 seconds later) This runs after 2 seconds
```

Benefits of Async Programming:

- Responsive applications - UI doesn't freeze
- Better performance - Multiple operations can run concurrently
- Improved user experience - No waiting for slow operations

Events - Quick Overview

What Are Events?
Events are actions or occurrences that happen in the system that your code can respond to.

Common Event Examples:

- User interactions: clicks, key presses, mouse movements
- Network events: data received, connection established
- File system events: file created, modified, deleted
- Timer events: timeout reached, interval triggered

Events - Quick Overview

Event-Driven Architecture:

- Event Emitters - Objects that emit events
- Event Listeners - Functions that respond to events
- Event Loop - Manages and processes events
- Non-blocking - Events don't stop other code from running

Why Events Matter:

- Decoupled code - Components communicate through events
- Reactive programming - Respond to changes as they happen
- Scalable applications - Handle many concurrent operations

Callbacks - Introduction

What is a Callback?

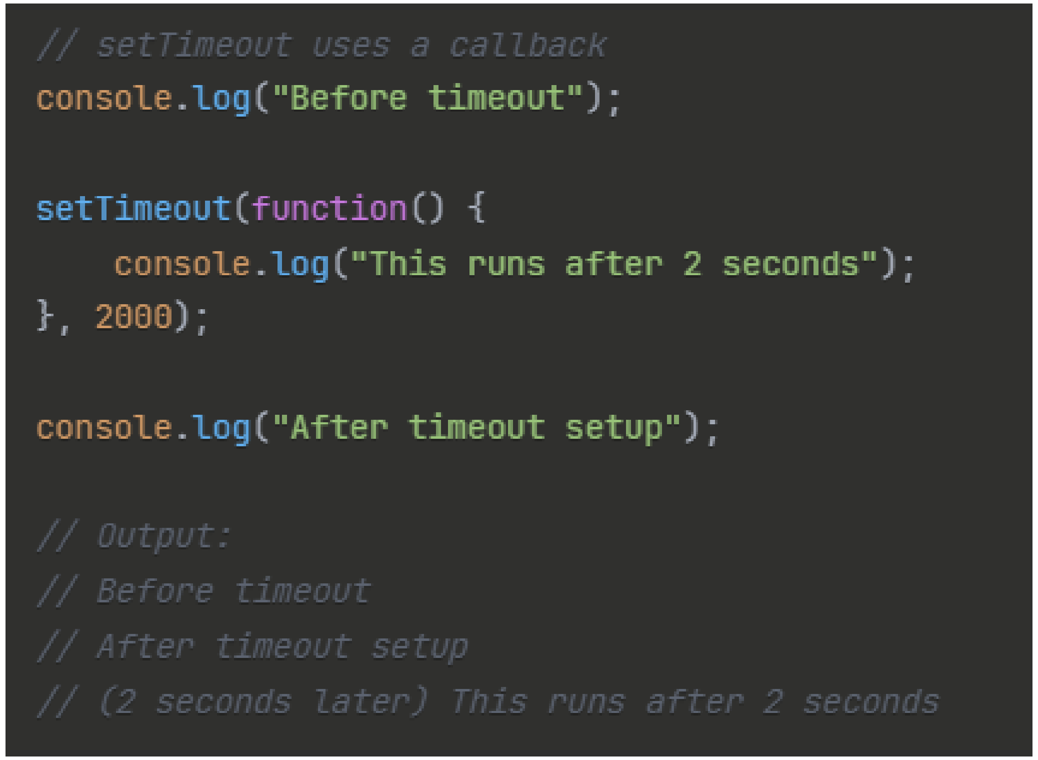
A callback is a function passed as an argument to another function, to be executed later.

Simple Callback Example:

```javascript
function greet(name, callback) {
    console.log(`Hello, ${name}!`);
    callback();
}

function afterGreeting() {
    console.log("Nice to meet you!");
}

greet("Alice", afterGreeting);
// Output:
// Hello, Alice!
// Nice to meet you!
```

Callbacks - Introduction

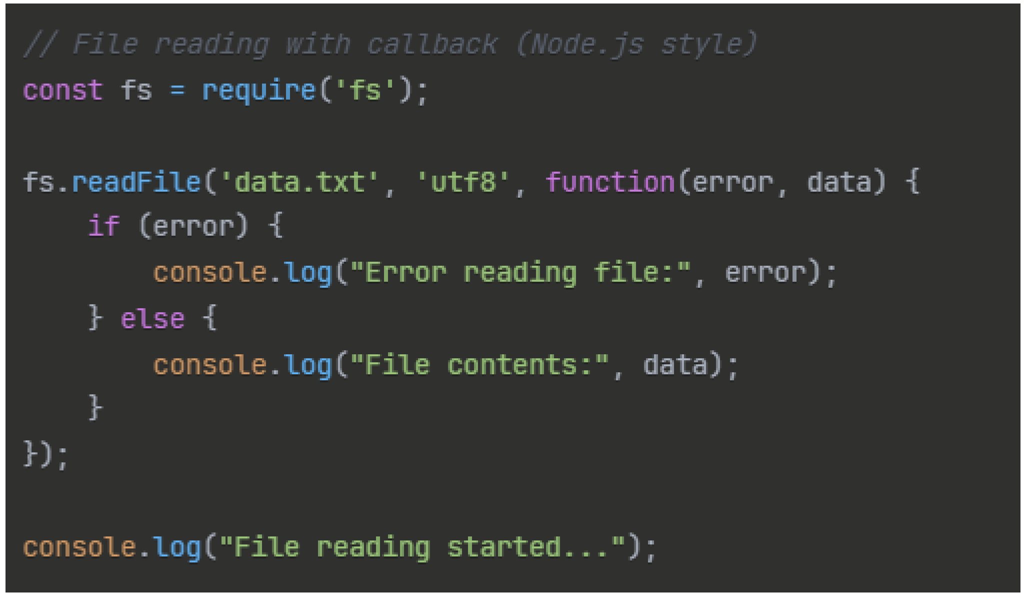
Asynchronous Callbacks:



```
// setTimeout uses a callback
console.log("Before timeout");

setTimeout(function() {
    console.log("This runs after 2 seconds");
}, 2000);

console.log("After timeout setup");

// Output:
// Before timeout
// After timeout setup
// (2 seconds later) This runs after 2 seconds
```

Callbacks - Introduction

Real-World Callback Example:



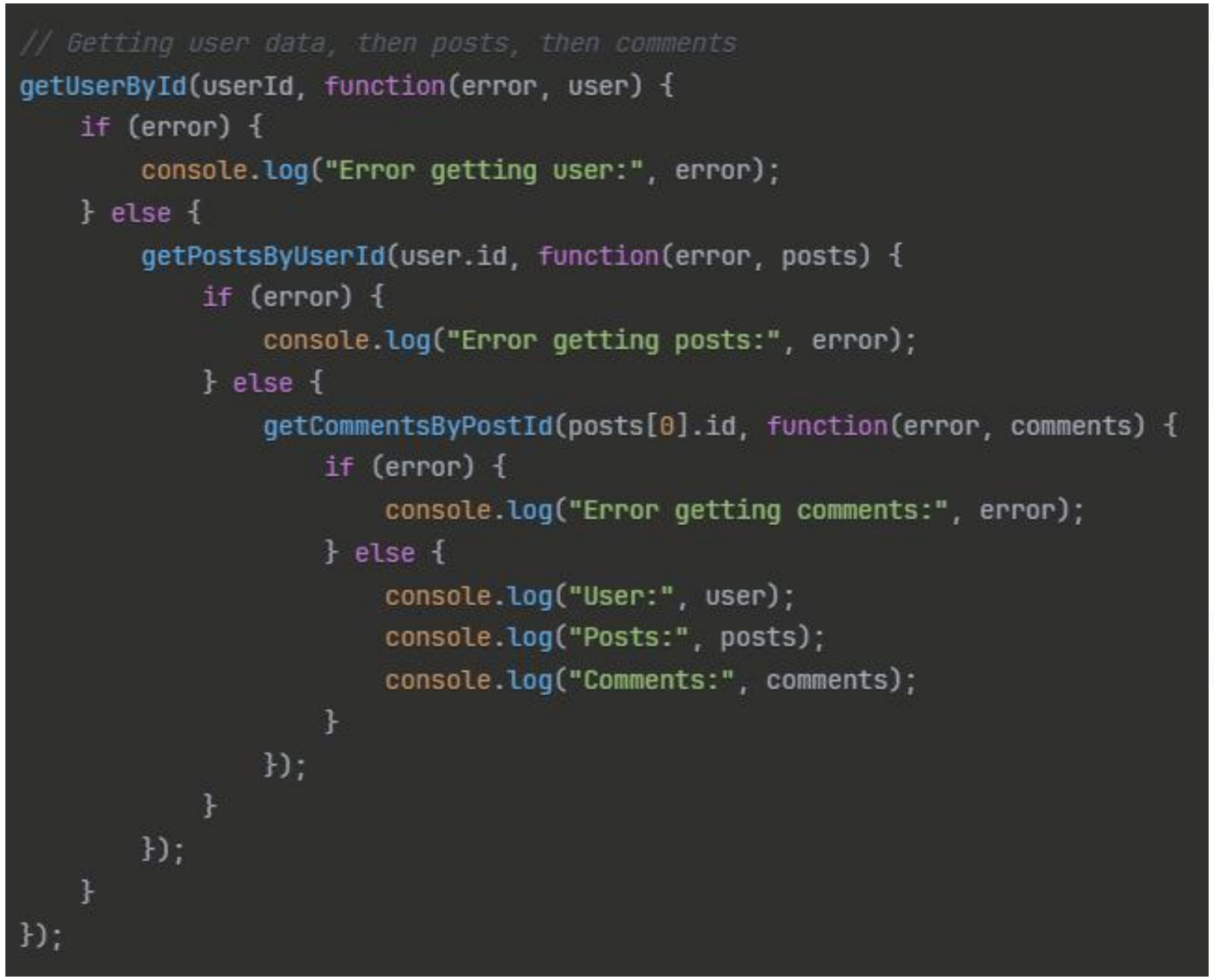
```javascript
// File reading with callback (Node.js style)
const fs = require('fs');

fs.readFile('data.txt', 'utf8', function(error, data) {
    if (error) {
        console.log("Error reading file:", error);
    } else {
        console.log("File contents:", data);
    }
});

console.log("File reading started...");
```

Callbacks - Introduction

Callback Challenges:

- Callback Hell - Nested callbacks become hard to read
- Error Handling - Need to handle errors in each callback
- Control Flow - Difficult to manage complex async operations

Callback Hell Problem



```
// Getting user data, then posts, then comments
getUserById(userId, function(error, user) {
    if (error) {
        console.log("Error getting user:", error);
    } else {
        getPostsByUserId(user.id, function(error, posts) {
            if (error) {
                console.log("Error getting posts:", error);
            } else {
                getCommentsByPostId(posts[0].id, function(error, comments) {
                    if (error) {
                        console.log("Error getting comments:", error);
                    } else {
                        console.log("User:", user);
                        console.log("Posts:", posts);
                        console.log("Comments:", comments);
                    }
                });
            }
        });
    }
});
```

Callback Hell Problem

Problems with This Approach:

- Hard to read - Code flows right instead of down
- Difficult to debug - Errors can occur at multiple levels
- Error handling duplication - Same error pattern repeated
- Maintenance nightmare - Adding features becomes complex

The Solution: Promises and Async/Await

Promises - Introduction

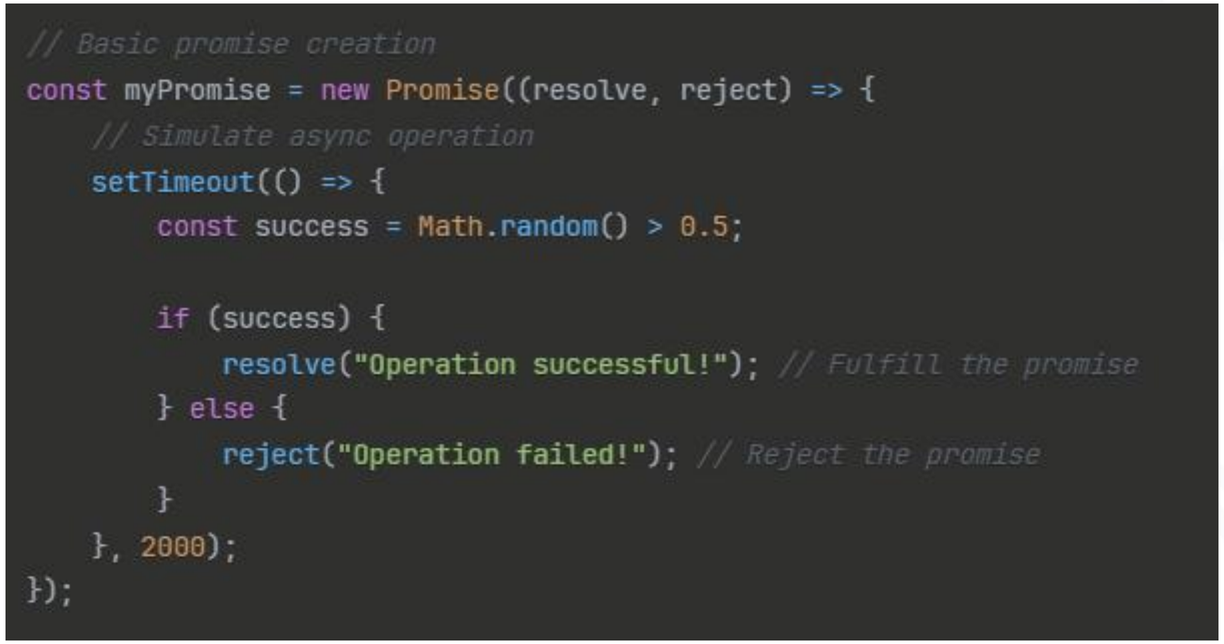
What is a Promise?
A Promise represents the eventual completion (or failure) of an asynchronous operation.

Promise States:

- Pending - Initial state, neither fulfilled nor rejected
- Fulfilled - Operation completed successfully
- Rejected - Operation failed

Promises - Introduction

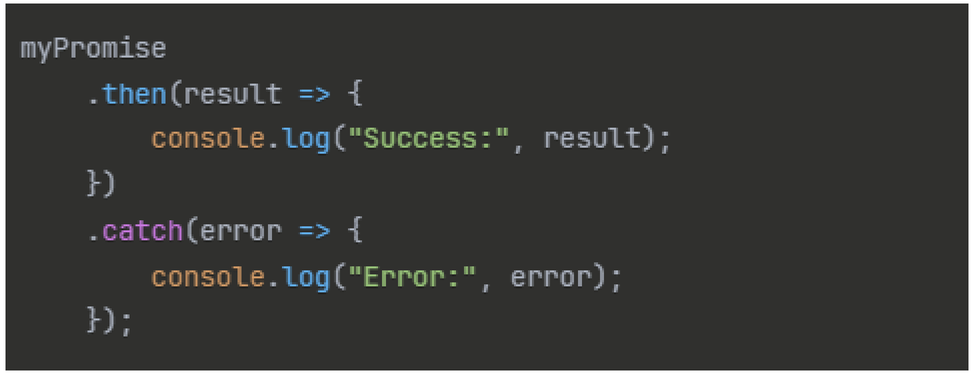
Creating a Promise:



```javascript
// Basic promise creation
const myPromise = new Promise((resolve, reject) => {
    // Simulate async operation
    setTimeout(() => {
        const success = Math.random() > 0.5;

        if (success) {
            resolve("Operation successful!"); // Fulfill the promise
        } else {
            reject("Operation failed!"); // Reject the promise
        }
    }, 2000);
});
```

Promises - Introduction

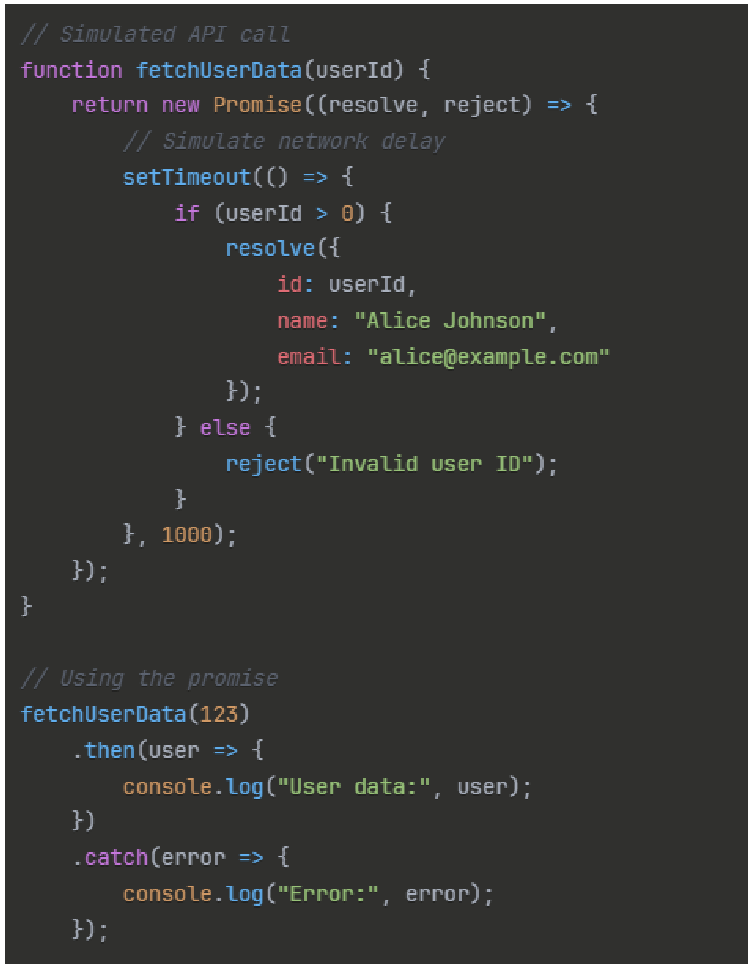
Using Promises with .then() and .catch():



```
myPromise
    .then(result => {
        console.log("Success:", result);
    })
    .catch(error => {
        console.log("Error:", error);
    });
```

Promises - Introduction

Real-World Promise Example:



```javascript
// Simulated API call
function fetchUserData(userId) {
    return new Promise((resolve, reject) => {
        // Simulate network delay
        setTimeout(() => {
            if (userId > 0) {
                resolve({
                    id: userId,
                    name: "Alice Johnson",
                    email: "alice@example.com"
                });
            } else {
                reject("Invalid user ID");
            }
        }, 1000);
    });
}

// Using the promise
fetchUserData(123)
    .then(user => {
        console.log("User data:", user);
    })
    .catch(error => {
        console.log("Error:", error);
    });
```

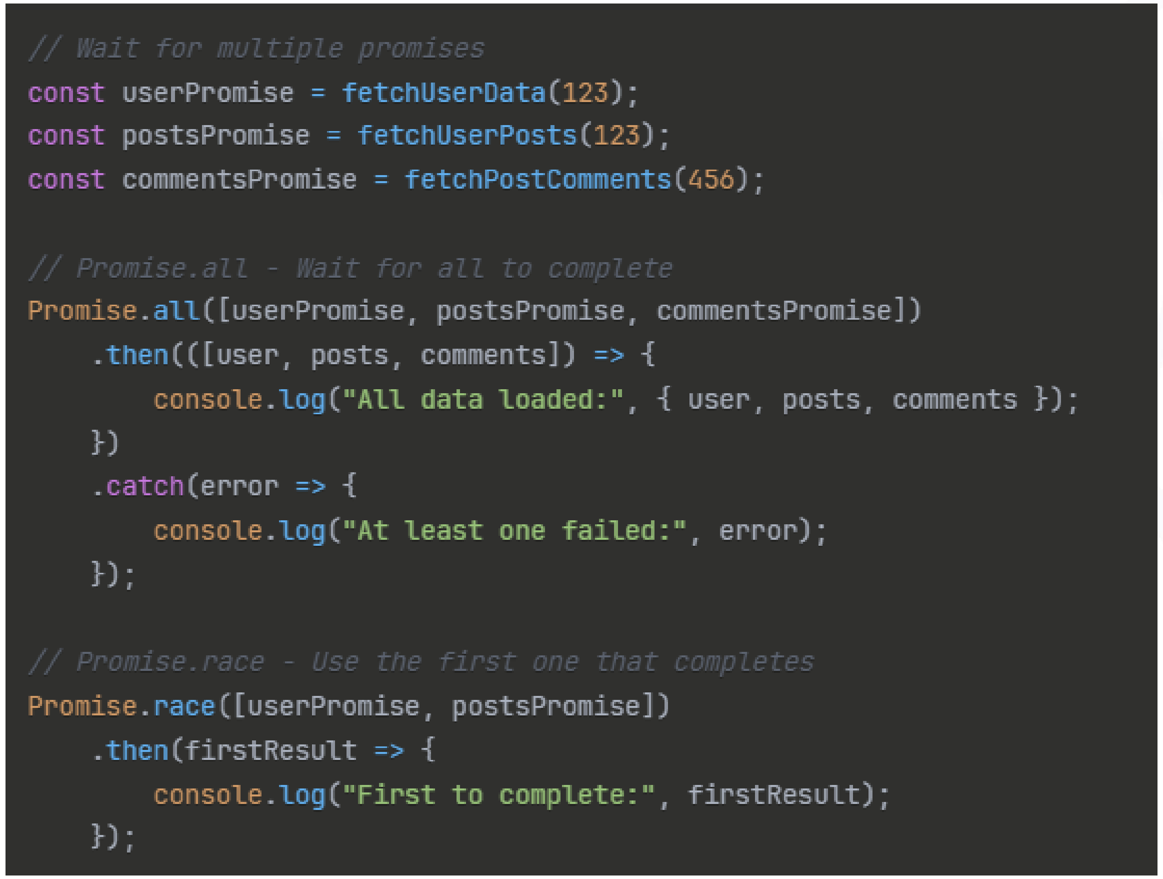
Promise Chaining

Chaining Promises:

```javascript
// Much cleaner than callback hell!
fetchUserData(123)
    .then(user => {
        console.log("Got user:", user);
        return fetchUserPosts(user.id); // Return another promise
    })
    .then(posts => {
        console.log("Got posts:", posts);
        return fetchPostComments(posts[0].id); // Return another promise
    })
    .then(comments => {
        console.log("Got comments:", comments);
    })
    .catch(error => {
        console.log("Error at any step:", error);
    });
```

Promise Chaining

Promise Helper Functions:



```javascript
// Wait for multiple promises
const userPromise = fetchUserData(123);
const postsPromise = fetchUserPosts(123);
const commentsPromise = fetchPostComments(456);

// Promise.all - Wait for all to complete
Promise.all([userPromise, postsPromise, commentsPromise])
    .then(([user, posts, comments]) => {
        console.log("All data loaded:", { user, posts, comments });
    })
    .catch(error => {
        console.log("At least one failed:", error);
    });

// Promise.race - Use the first one that completes
Promise.race([userPromise, postsPromise])
    .then(firstResult => {
        console.log("First to complete:", firstResult);
    });
```

Promise Chaining

Benefits of Promises:

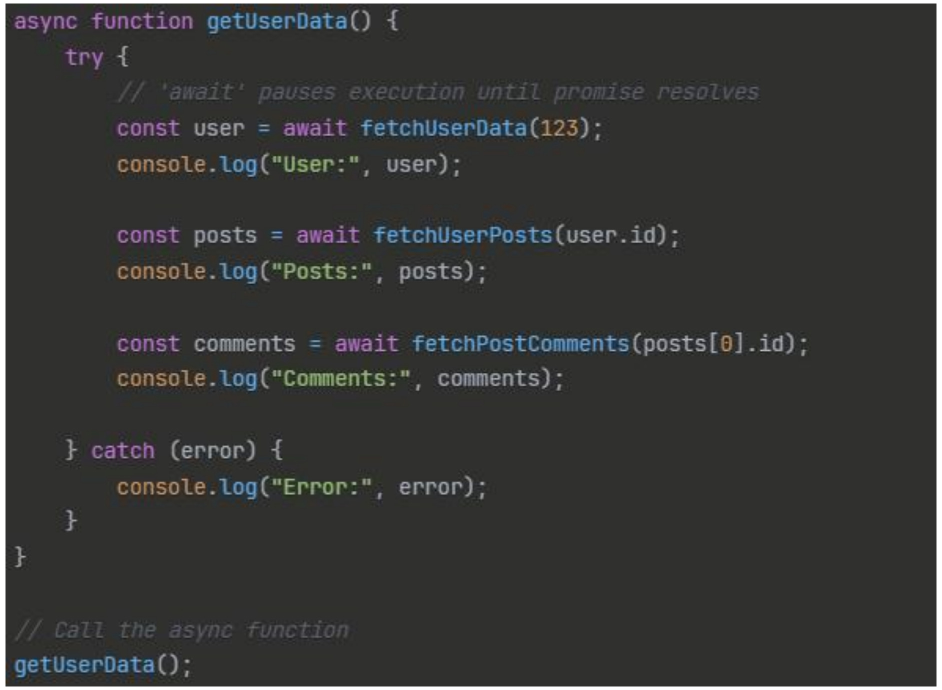
- Cleaner syntax - No more callback hell
- Better error handling - Single .catch() for all errors
- Composable - Easy to combine multiple async operations
- Readable - Code flows top to bottom

Async/Await - Modern Syntax

What is Async/Await?

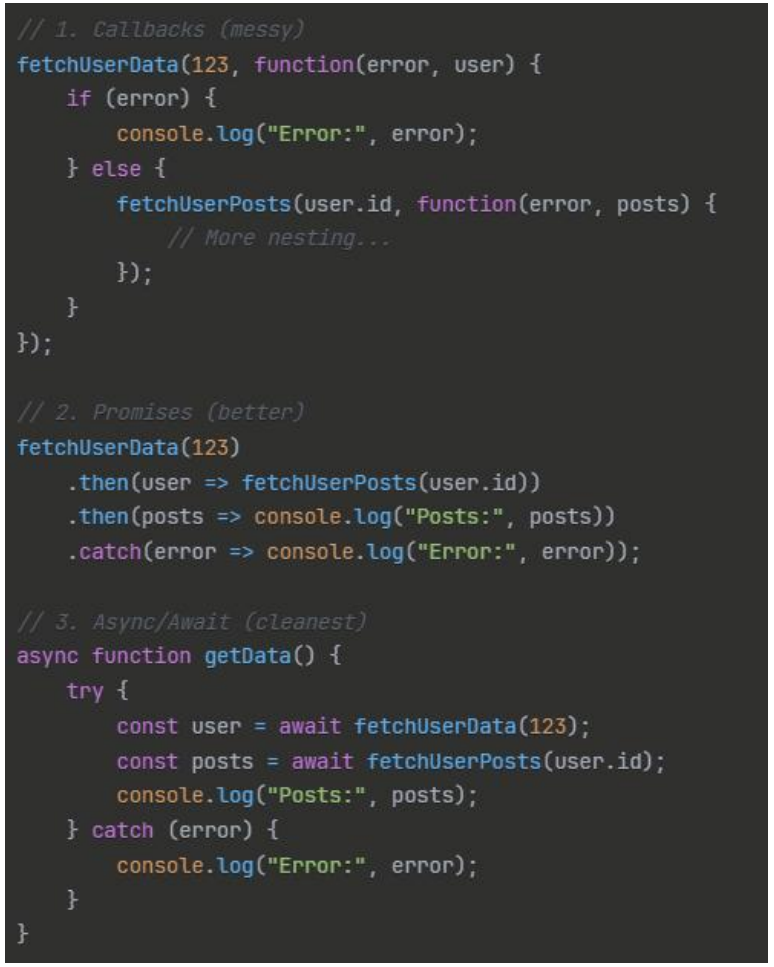
Async/Await is syntactic sugar over Promises that makes asynchronous code look and feel like synchronous code

Basic Async/Await Syntax:



```
async function getUserData() {
    try {
        // 'await' pauses execution until promise resolves
        const user = await fetchUserData(123);
        console.log("User:", user);

        const posts = await fetchUserPosts(user.id);
        console.log("Posts:", posts);

        const comments = await fetchPostComments(posts[0].id);
        console.log("Comments:", comments);

    } catch (error) {
        console.log("Error:", error);
    }
}

// Call the async function
getUserData();
```

Comparing All Three Approaches:



```
// 1. Callbacks (messy)
fetchUserData(123, function(error, user) {
    if (error) {
        console.log("Error:", error);
    } else {
        fetchUserPosts(user.id, function(error, posts) {
            // More nesting...
        });
    }
});

// 2. Promises (better)
fetchUserData(123)
    .then(user => fetchUserPosts(user.id))
    .then(posts => console.log("Posts:", posts))
    .catch(error => console.log("Error:", error));

// 3. Async/Await (cleanest)
async function getData() {
    try {
        const user = await fetchUserData(123);
        const posts = await fetchUserPosts(user.id);
        console.log("Posts:", posts);
    } catch (error) {
        console.log("Error:", error);
    }
}
```

Working with APIs - Fetch Introduction

What is an API?

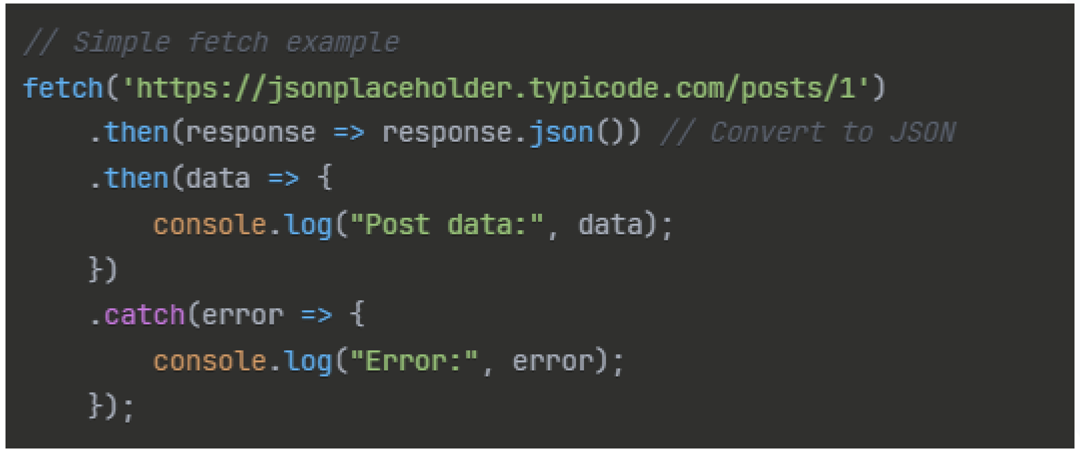
- Application Programming Interface
- Communication contract between different software systems
- HTTP APIs use standard web protocols (GET, POST, PUT, DELETE)
- JSON is the most common data format

The Fetch API:

Built-in browser function for making HTTP requests (also available in Node.js 18+).

Working with APIs - Fetch Introduction

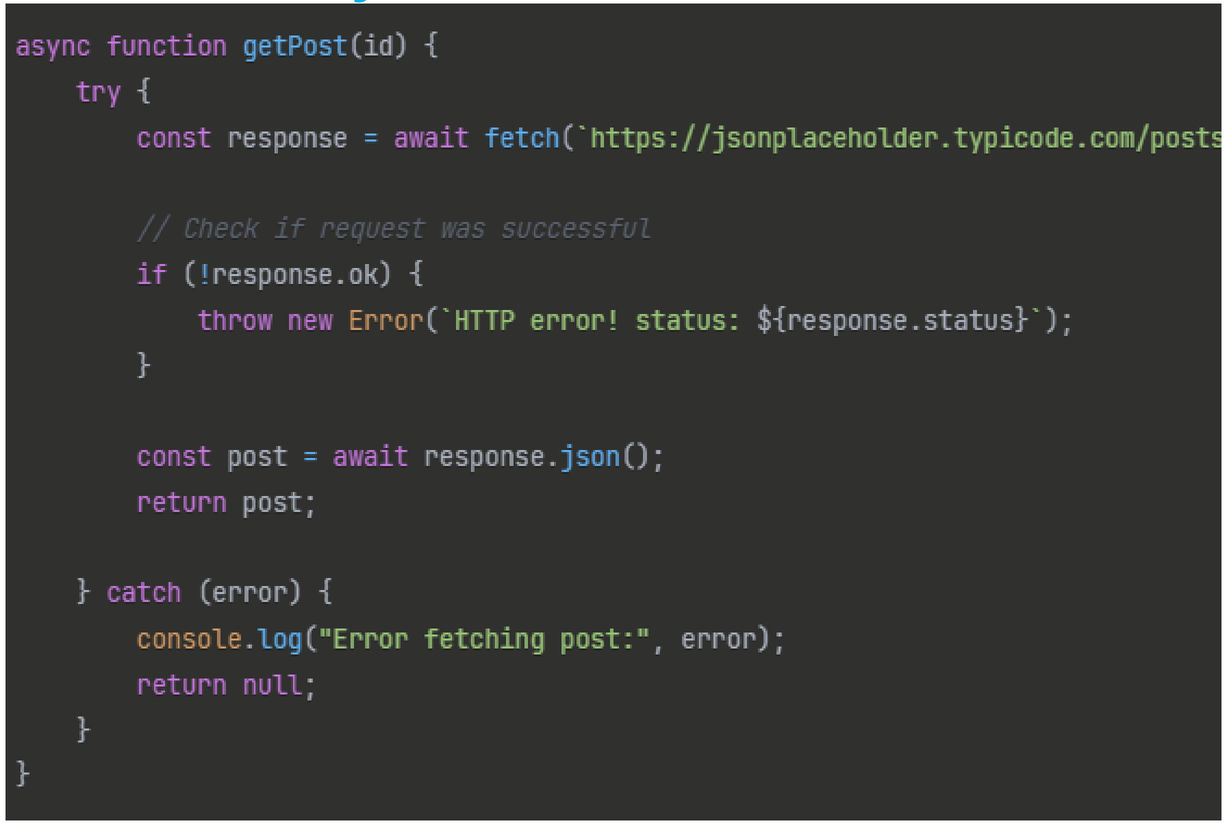
Basic GET Request:



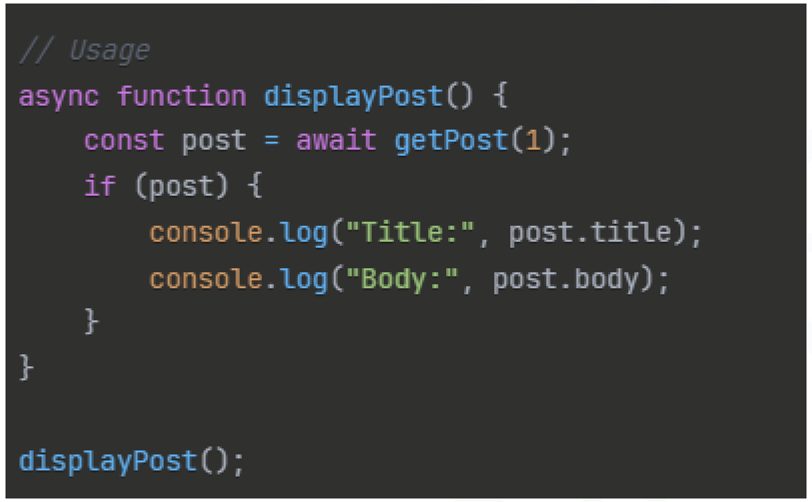
```javascript
// Simple fetch example
fetch('https://jsonplaceholder.typicode.com/posts/1')
    .then(response => response.json()) // Convert to JSON
    .then(data => {
        console.log("Post data:", data);
    })
    .catch(error => {
        console.log("Error:", error);
    });
```

Working with APIs - Fetch Introduction

Fetch with Async/Await:



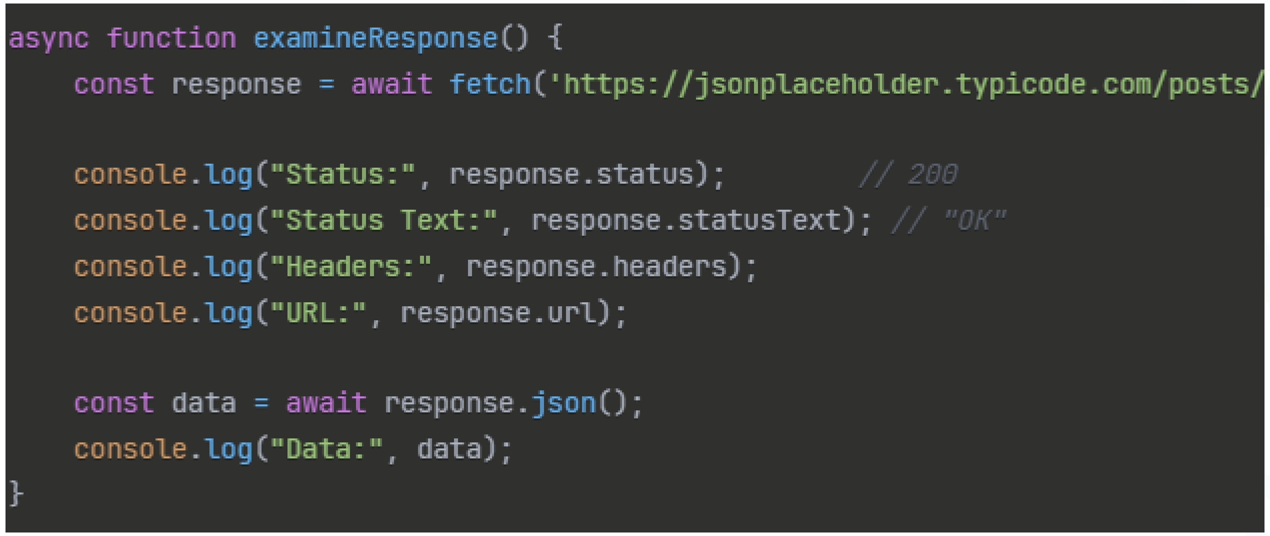
```
async function getPost(id) {
    try {
        const response = await fetch(`https://jsonplaceholder.typicode.com/posts

        // Check if request was successful
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }

        const post = await response.json();
        return post;

    } catch (error) {
        console.log("Error fetching post:", error);
        return null;
    }
}
```



```
// Usage
async function displayPost() {
    const post = await getPost(1);
    if (post) {
        console.log("Title:", post.title);
        console.log("Body:", post.body);
    }
}

displayPost();
```

Working with APIs - Fetch Introduction

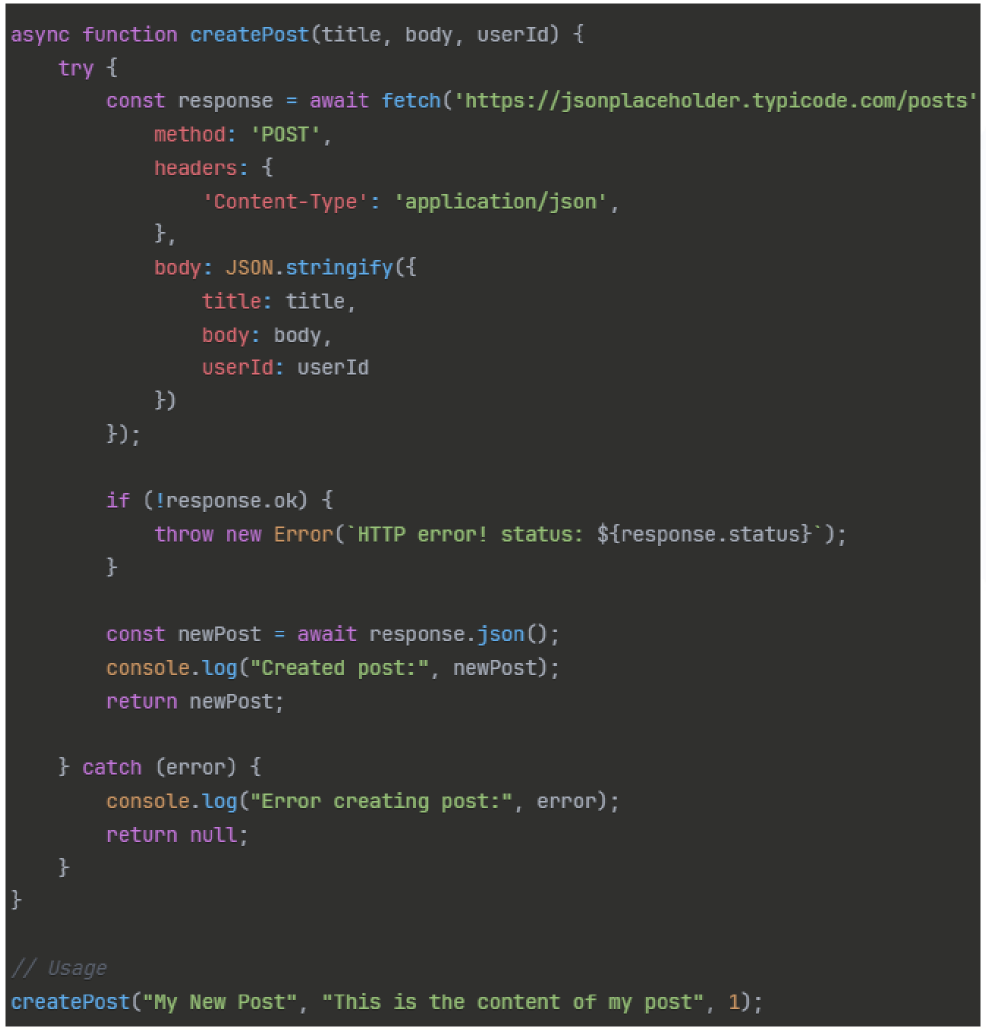
Understanding HTTP Response:



```
async function examineResponse() {
    const response = await fetch('https://jsonplaceholder.typicode.com/posts/

    console.log("Status:", response.status);        // 200
    console.log("Status Text:", response.statusText); // "OK"
    console.log("Headers:", response.headers);
    console.log("URL:", response.url);

    const data = await response.json();
    console.log("Data:", data);
}
```

Making POST Requests

POST Request with Fetch:



```javascript
async function createPost(title, body, userId) {
    try {
        const response = await fetch('https://jsonplaceholder.typicode.com/posts'
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({
                title: title,
                body: body,
                userId: userId
            })
        });

        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }

        const newPost = await response.json();
        console.log("Created post:", newPost);
        return newPost;

    } catch (error) {
        console.log("Error creating post:", error);
        return null;
    }
}

// Usage
createPost("My New Post", "This is the content of my post", 1);
```

Making POST Requests

Other HTTP Methods:

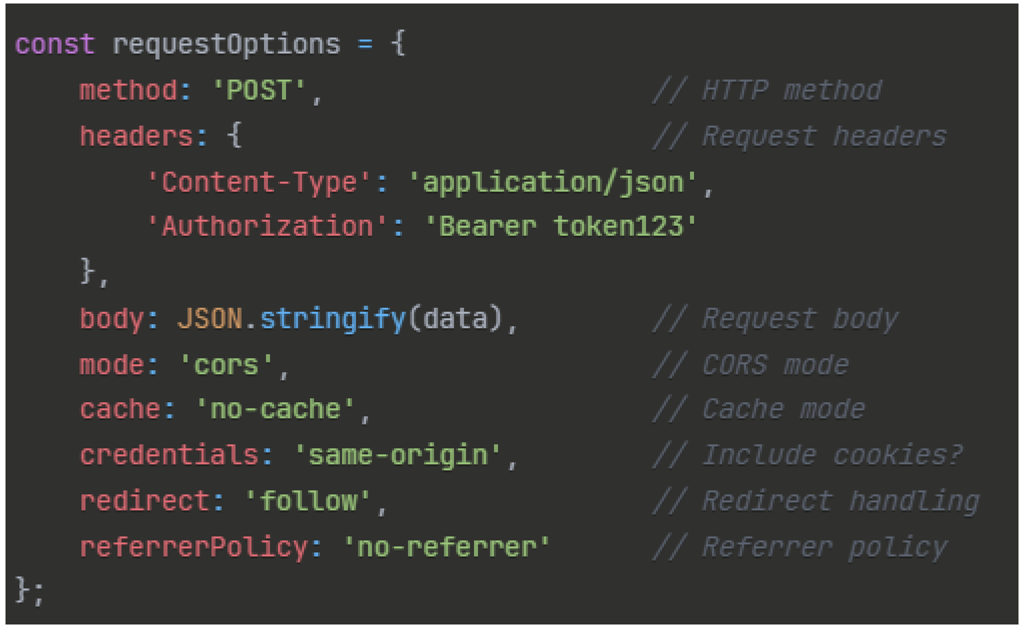
```javascript
// PUT - Update entire resource
async function updatePost(id, title, body, userId) {
    const response = await fetch(`https://jsonplaceholder.typicode.com/posts/${id
        method: 'PUT',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ id, title, body, userId })
    });
    return response.json();
}
```

```javascript
// PATCH - Update partial resource
async function patchPost(id, updates) {
    const response = await fetch(`https://jsonplaceholder.typicode.com/posts/${id}
        method: 'PATCH',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(updates)
    });
    return response.json();
}
```

```javascript
// DELETE - Remove resource
async function deletePost(id) {
    const response = await fetch(`https://jsonplaceholder.typicode.com/posts/${id}
        method: 'DELETE'
    });
    return response.ok;
}
```

Making POST Requests

Request Configuration Options:



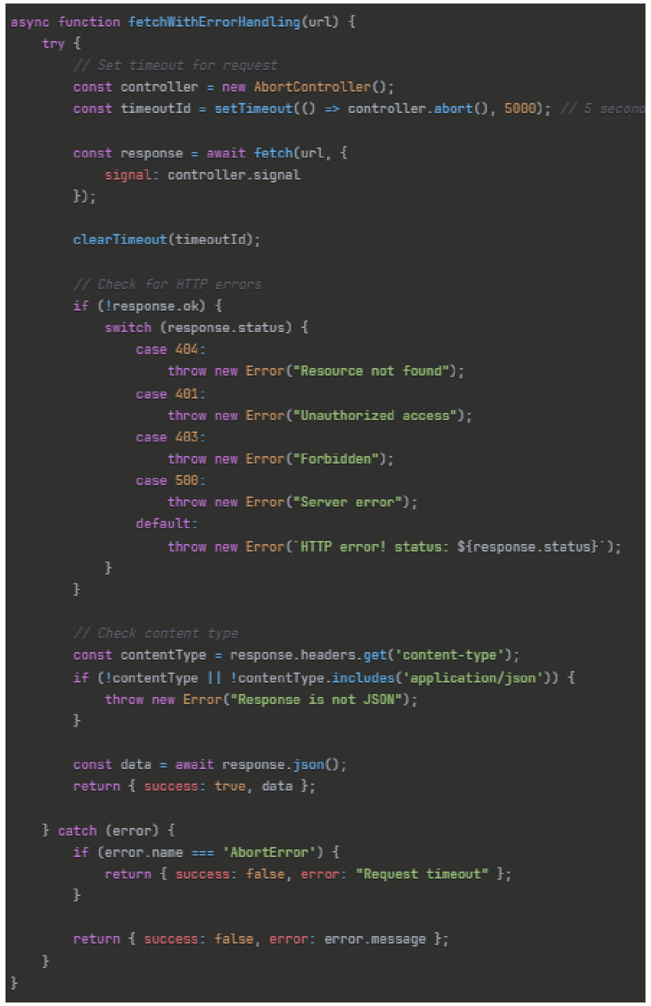
```javascript
const requestOptions = {
    method: 'POST',                     // HTTP method
    headers: {                          // Request headers
        'Content-Type': 'application/json',
        'Authorization': 'Bearer token123'
    },
    body: JSON.stringify(data),         // Request body
    mode: 'cors',                       // CORS mode
    cache: 'no-cache',                  // Cache mode
    credentials: 'same-origin',         // Include cookies?
    redirect: 'follow',                 // Redirect handling
    referrerPolicy: 'no-referrer'       // Referrer policy
};
```

Error Handling with APIs

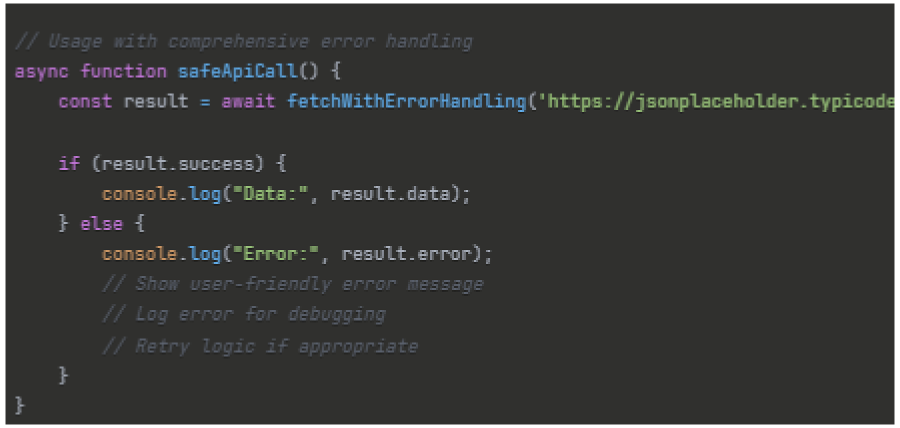
Types of API Errors:

1. Network errors - No internet, server down
2. HTTP errors - 404 Not Found, 500 Server Error
3. Parsing errors - Invalid JSON response
4. Timeout errors - Request takes too long

Error Handling with APIs



```javascript
async function fetchWithErrorHandling(url) {
    try {
        // Set timeout for request
        const controller = new AbortController();
        const timeoutId = setTimeout(() => controller.abort(), 5000); // 5 second

        const response = await fetch(url, {
            signal: controller.signal
        });

        clearTimeout(timeoutId);

        // Check for HTTP errors
        if (!response.ok) {
            switch (response.status) {
                case 404:
                    throw new Error("Resource not found");
                case 401:
                    throw new Error("Unauthorized access");
                case 403:
                    throw new Error("Forbidden");
                case 500:
                    throw new Error("Server error");
                default:
                    throw new Error(`HTTP error! status: ${response.status}`);
            }
        }

        // Check content type
        const contentType = response.headers.get('content-type');
        if (!contentType || !contentType.includes('application/json')) {
            throw new Error("Response is not JSON");
        }

        const data = await response.json();
        return { success: true, data };

    } catch (error) {
        if (error.name === 'AbortError') {
            return { success: false, error: "Request timeout" };
        }

        return { success: false, error: error.message };
    }
}
```



```javascript
// Usage with comprehensive error handling
async function safeApiCall() {
    const result = await fetchWithErrorHandling('https://jsonplaceholder.typicode

    if (result.success) {
        console.log("Data:", result.data);
    } else {
        console.log("Error:", result.error);
        // Show user-friendly error message
        // Log error for debugging
        // Retry logic if appropriate
    }
}
```

Practice: JSONPlaceholder API Integration

Your Task:
Use the JSONPlaceholder API to fetch and display a list of posts

Requirements:

- Use the JSONPlaceholder API: https://jsonplaceholder.typicode.com/posts
- Fetch all posts using async/awaitDisplay each post's title and body
- Add error handling for network issues
- Create a function to fetch a single post by ID
- Bonus: Add a function to create a new post
- Use either Fetch API or Axios (your choice)

THANK YOU

ADAEGY