**Promise Chaining**

**Promise chaining** allows you to perform a series of asynchronous operations in sequence. Each .then() call returns a new promise, allowing you to chain multiple asynchronous operations. This makes it easier to manage complex asynchronous workflows by avoiding deeply nested callbacks.

Here's a simple example of promise chaining:

fetch('https://api.example.com/data')

.then(response => {

if (!response.ok) {

throw new Error('Network response was not ok');

}

return response.json(); // This returns a promise

})

.then(data => {

// Process the JSON data

console.log('Data received:', data);

return data.someProperty; // Return a value to pass to the next .then()

})

.then(someValue => {

// Do something with the value

console.log('Processed value:', someValue);

})

.catch(error => {

// Handle any error that occurred in the chain

console.error('Error occurred:', error);

});

In this example:

1. **fetch()** starts the chain with a promise that resolves to a Response object.
2. **First .then()** checks if the response is ok, parses it to JSON, and returns that JSON, which itself is a promise.
3. **Second .then()** receives the parsed JSON data and performs additional processing.
4. **Third .then()** works with the processed data.
5. **.catch()** at the end of the chain catches any errors that occurred at any stage of the chain.

**Async/Await with Error Handling**

**async/await** provides a more readable and synchronous-looking way to handle asynchronous code compared to promise chaining. It is syntactic sugar over promises, allowing you to write asynchronous code in a more synchronous style, which can be easier to read and maintain.

**Basic Syntax**

* **async**: Used to declare a function that returns a promise. The function automatically wraps its return value in a promise.
* **await**: Used to wait for a promise to resolve or reject. It can only be used inside async functions.

Here’s a basic example:

async function fetchData() {

try {

const response = await fetch('https://api.example.com/data');

if (!response.ok) {

throw new Error('Network response was not ok');

}

const data = await response.json();

console.log('Data received:', data);

// Process data

const processedValue = data.someProperty;

console.log('Processed value:', processedValue);

return processedValue; // This will be the resolved value of the promise returned by fetchData

} catch (error) {

console.error('Error occurred:', error);

// Handle errors here, possibly rethrow or return a default value

}

}

fetchData();

**Explanation:**

1. **async function fetchData()**: Declares fetchData as an asynchronous function, which implicitly returns a promise.
2. **const response = await fetch('https://api.example.com/data');**: Waits for the fetch promise to resolve. This is similar to the first .then() in promise chaining.
3. **if (!response.ok)**: Checks if the response was successful; if not, throws an error.
4. **const data = await response.json();**: Waits for the JSON parsing to complete. This is similar to the second .then() in promise chaining.
5. **Processing and logging data**: Continues with data processing, similar to further .then() calls.
6. **catch (error)**: Catches any errors that occurred in the try block, providing a way to handle errors, similar to .catch() in promise chaining.

**Advanced Example with Multiple Async Operations**

Here’s how you might handle multiple asynchronous operations with async/await:

async function processMultipleRequests() {

try {

const [response1, response2] = await Promise.all([

fetch('https://api.example.com/data1'),

fetch('https://api.example.com/data2')

]);

if (!response1.ok || !response2.ok) {

throw new Error('One or more responses were not ok');

}

const [data1, data2] = await Promise.all([

response1.json(),

response2.json()

]);

console.log('Data 1:', data1);

console.log('Data 2:', data2);

// Process the data

// ...

} catch (error) {

console.error('Error occurred:', error);

}

}

processMultipleRequests();

In this example:

* **Promise.all()**: Used to wait for multiple fetch requests to complete in parallel.
* **await Promise.all([...])**: Awaits the resolution of all promises in the array.
* **Error handling**: Catches any errors that occur in any part of the try block, including issues with either of the fetch requests or JSON parsing.

By using async/await, you can write asynchronous code that looks more like traditional synchronous code, making it easier to understand and manage, especially when dealing with complex logic and multiple asynchronous operations.

Top of Form

Bottom of Form