Asynchronous JavaScript and Error Handling

David Tucker

CTO Consultant

@_davidtucker_ | davidtucker.net

Asynchronous Use Cases

- Asking for user input
- Calling an API
- Reading and writing files to the filesystem
- Uploading and downloading files
- Interacting with a database
- Interacting with an external service

Using Callbacks

index.js

```
// Importing the Node filesystem module
import fs from 'node:fs';

// Callback function
const fileWriteCompleted = () => {
  console.log("File written");
};

// Writing the data to the file
const data = JSON.stringify({});
fs.writeFile(                                    );
```

Pyramid of Doom

index.js

```
// Importing the Node filesystem module
import fs from 'node:fs';

// Writing the data to the file
const data = JSON.stringify({});
fs.writeFile("./file1.json", data, () => {
  console.log("File written");
});
```

Promise

The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

Using Promises

index.js

```
// Importing the Node filesystem module for promises
import fs from 'node:fs/promises';

// Writing the data to the file
const data = JSON.stringify({});
fs.writeFile("./file1.json", data)
   .then(() => {
     return fs.writeFile("./file2.json", data)
   })
   .then(() => {
     return fs.writeFile("./file3.json", data)
   })
   .then(() => {
     return fs.writeFile("./file4.json", data)
   });
```

Using Promises

index.js

```
// Importing the Node filesystem module for promises
import fs from 'node:fs/promises';

// Writing the data to the file
const data = JSON.stringify({});
fs.writeFile("./file1.json", data)
   .then(() => fs.writeFile("./file2.json", data))
   .then(() => fs.writeFile("./file3.json", data))
   .then(() => fs.writeFile("./file4.json", data));
```

Async and Await

index.js

```
// Importing the Node filesystem module for promises
import fs from 'node:fs/promises';

// Data to be written
const data = JSON.stringify({});

// Using Promises (without async/await)
fs.writeFile("./file1.json", data)
  .then(() => {
    console.log("File written");
  });

// Using Async/Await
await fs.writeFile("./file1.json", data);
console.log("File written");
```

Chaining Async and Await

index.js

```
// Importing the Node filesystem module for promises
import fs from 'node:fs/promises';

// Data to be written
const data = JSON.stringify({});

// Using Async/Await
await fs.writeFile("./file1.json", data);
await fs.writeFile("./file2.json", data);
await fs.writeFile("./file3.json", data);
await fs.writeFile("./file4.json", data);
console.log("Files written");
```

Async Functions

index.js

```
// Importing the Node filesystem module for promises
import fs from 'node:fs/promises';

// Data to be written
const data = JSON.stringify({});

// Function marked as async
async function writeFiles() {
  await fs.writeFile("./file1.json", data);
  await fs.writeFile("./file2.json", data);
  await fs.writeFile("./file3.json", data);
  await fs.writeFile("./file4.json", data);
}

// Execute async function
writeFiles().then(() => console.log("Files written"));
```

Error Handling

Without Error Handling

index.js

```
// Importing the Node filesystem module
import fs from 'node:fs/promises';

// Reading a file and parsing it as JSON
const rawData = await fs.readFile("./data.json");
const dataObj = JSON.parse(rawData);
```

Using Try Catch

index.js

```
// Importing the Node filesystem module
import fs from 'node:fs/promises';

// Reading a file and parsing it as JSON
try {


} catch (err) {


}
```

Without Async Await

index.js

```
// Importing the Node filesystem module
import fs from 'node:fs/promises';

// Reading a file and parsing it as JSON
fs.readFile("./data.json")
  .then(rawData => {
    return JSON.parse(rawData);
  })
  .catch(err => {


  });
```

Using Callbacks

Throw

The throw statement throws a user-defined exception. Execution of the current function will stop (the statements after throw won't be executed), and control will be passed to the first catch block in the call stack. If no catch block exists among caller functions, the program will terminate.

Using Promises

Using Promises with Async and Await

Async and Await

The async function declaration declares an async function where the await keyword is permitted within the function body. The async and await keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains... Use of async and await enables the use of ordinary try / catch blocks around asynchronous code.

Top Level Await

Top-level await enables modules to act as big async functions: With top-level await, ECMAScript Modules (ESM) can await resources, causing other modules who import them to wait before they start evaluating their body.

Integrating Asynchronous Concepts

Demo

Loading data using a file on the local file system

Utilizing asynchronous logic with promises and async/await

Persisting data beyond a single execution of the application

Making HTTP Requests

Demo

- Reviewing a public API for currency conversion
- Signing up for a free API key for the API
- Testing the API with Postman
- Implementing the API in JavaScript with fetch

Integrating Currency Data

Demo

Integrating currency data from the public API

Adding additional prompts for salary fields

Displaying formatted currency fields

Verifying data persistence

Concurrent Promises

Event Handling

Events

Events are things that happen in the system you are programming — the system produces (or "fires") a signal of some kind when an event occurs, and provides a mechanism by which an action can be automatically taken (that is, some code running) when the event occurs.

Event Handling Use Cases

System events

Mouse clicks

Key presses

Async progress events