Async-Await

In JavaScript, the async and await keywords are used to work with Promises more easily. They were introduced in ECMAScript 2017 (ES8) to make asynchronous code look and behave more like synchronous code, making it easier to read and understand.

When we perform asynchronous operations (like fetching data from an API or reading files), JavaScript does not wait for these operations to complete. It continues executing the remaining code and then returns the result when the asynchronous operation is complete. To handle this behaviour, JavaScript originally used callbacks and later introduced Promises to make handling asynchronous code more manageable.

The **problem with callbacks and promises** is that they can lead to nested code (callback hell) or long chains of .then() calls, which can be hard to read and understand. The async/await syntax was designed to solve these issues by making asynchronous code look more like synchronous code.

async function

- An async function is a special type of function that always returns a Promise, regardless of what you explicitly return from it.
- If you return a value from an async function, it is automatically wrapped in a resolved promise.
- If you throw an error, it is automatically wrapped in a rejected promise.

Syntax of async function

```
async function myFunction() {
  return "value";
}
```

Example

```
async function myFunction() {
        return "Welcome to Learn2Earn Labs Training Institute, Agra";
}
myFunction().then(result => console.log(result));
```

In the example, the async function (myFunction), returns the string. However, under the hood, myFunction actually returns a resolved promise with the string value.

await

- await is used to pause the execution of an async function until the promise is resolved or rejected.
- It allows you to wait for an asynchronous operation to complete before moving on to the next line of code.
- The keyword await can only be used inside an async function.
- When you use await, JavaScript waits for the promise to resolve, and then continues execution of the async function.
- If the promise resolves, await returns the resolved value.
- If the promise is rejected, await throws an error, which can be caught using try/catch.

Example

```
async function myAsyncFunction() {
  let promise = new Promise((resolve, reject) => {
                    setTimeout(() => resolve("Hello Students!"), 1000);
  });

  let result = await promise;  // pausing execution & Waits until the promise resolves
  console.log(result);         // Prints "Hello Students!" after 1 second
}
myAsyncFunction();
```

Why do we use await inside async functions?

- **Code Readability:** Using await makes your asynchronous code look like synchronous code, making it more readable and easier to follow. Without await, you would need to chain .then() calls or use nested callbacks.
- **Error Handling:** With await, error handling can be done using try/catch blocks, which are easier to manage compared to .catch() with promises. This makes it simpler to handle both synchronous and asynchronous errors in the same way.
- **Avoiding Callback Hell:** The old way of handling asynchronous code using callbacks often led to callback hell, where functions are deeply nested inside one another. async/await flattens the code and avoids such deeply nested structures.

How async/await works under the hood

Internally, async functions still rely on Promises. When an async function is called, it returns a promise. The await expression causes the function execution to pause, and JavaScript waits for the promise to settle (either fulfilled or rejected) before proceeding.

Async/await process step by step:

- When you call an async function, it immediately returns a promise.
- The execution of the async function pauses when it encounters an await expression and waits for the promise to be resolved or rejected.
- Once the promise is resolved, the function resumes execution with the resolved value.
- If the promise is rejected, the function throws the error, which can be caught using try/catch.

Example

```
function readFile(fileName) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (fileName === "myfile.txt") {
        resolve("You are too good");
      } else {
        reject("Error: File not found");
      }
    }, 2000);
  });  }
async function processFile() {
  try {
    let data = await readFile("myfile.txt");  // pausing execution & simulate reading
                                                a valid file
    console.log("The data is : "+data);
  } catch (error) {
    console.error(error);  // Handle any errors
  }
}
processFile();
```

Examples of returning promises from async functions

Example

```
async function getUserDetails(userId) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (userId === 1) {
        resolve({
                        id: 1,
                        name: "Neha Rathore"
                });
      } else {
        reject("User not found");
      }
    }, 1000);
  });
}

getUserDetails(1).then(user => console.log(user)).catch(error =>
console.error(error));
```

Example

```
async function generateMessage() {
  return new Promise((resolve) => {
    setTimeout(() => {
                        resolve("God Bless You");
                         }, 500);
  });
}

generateMessage().then(message => console.log("Generated Message :",
message));
```

Awaiting Promises and non-promises

Example: Awaiting a Promise

In the example, the await keyword pauses the execution until the promise resolves.

```
async function execute() {
  let promise = new Promise((resolve) => {
    setTimeout(() => resolve("Promise Resolved!"), 1000);
  });
  let result = await promise;  // Pauses until the promise resolves
  console.log(result);         // Outputs: "Promise Resolved!"
}
execute();
```

Example: Awaiting a Non-promise (Synchronous Value)

You can await a regular value (not a promise). In this case, await does not pause execution and just returns the value immediately.

```
async function execute() {
    let result = await "Hello Students";  // No need to pause, returns immediately
    console.log(result);    // Outputs: Hello Students
  }
  execute();
```

Example: Awaiting a Function that Returns a Promise

In the example, await is used to wait for a function that returns a promise.

```
function getPromise() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Promise Returned!"), 1500);
  });}
async function execute() {
  let result = await getPromise();  // Pauses until the promise resolves
  console.log(result);              // Outputs: "Promise Returned!"
}
execute();
```

Example: Awaiting Multiple Promises with Promise.all

You can use await with Promise.all to wait for multiple promises to resolve.

```
async function execute() {
    let promise1 = new Promise((resolve) => setTimeout(() => resolve("First
    Promise resolved"), 1000));

    let promise2 = new Promise((resolve) => setTimeout(() =>
    resolve("Second Promise resolved"), 2000));

    let results = await Promise.all([promise1, promise2]);  // Pauses until both
                                                              promises resolve
    console.log(results);
}
execute();
```

Example: Awaiting in a Loop

You can use await inside loops to wait for asynchronous operations in each iteration.

```
async function execute() {
  let numbers = [10, 20, 30, 40, 50];

  for (let num of numbers) {
    let result = await new Promise((resolve) => {
      setTimeout(() => resolve(num/2), 1000);
    });
    console.log(result);
  }
}

execute();
```

Handling multiple await statements

Example: Sequential await Statements

In the example, each await statement is executed sequentially, meaning the code waits for the first operation to complete before starting the next one.

```
async function execute() {

let p1 = await new Promise((resolve) => setTimeout(() => resolve("First
Promise"), 1000));
  console.log(p1);

let p2 = await new Promise((resolve) => setTimeout(() => resolve("Second
Promise"), 2000));
  console.log(p2);

let p3 = await new Promise((resolve) => setTimeout(() => resolve("Third
Promise"), 5000));
  console.log(p3);
}
execute();
```

Example: Parallel Execution Using Promise.all

To execute multiple asynchronous operations in parallel (instead of sequentially), you can use Promise.all to wait for all the promises at once.

```
async function execute() {
  let promises = [
    new Promise((resolve) => setTimeout(() => resolve("First Result"), 1000)),
    new Promise((resolve) => setTimeout(() => resolve("Second Result"), 2000)),
    new Promise((resolve) => setTimeout(() => resolve("Third Result"), 3000))
  ];
  let results = await Promise.all(promises);
  console.log(results);
}
execute();
```

Example: Handling Multiple await with try/catch for Error Handling

You can handle errors by wrapping multiple await statements in try/catch blocks to catch any rejected promises.

```
async function execute() {
  try {
        let p1 = await new Promise((resolve) => setTimeout(() => resolve("First
        Promise"), 1000));
        console.log(p1);

        let p2 = await new Promise((resolve, reject) => setTimeout(() =>
        reject("Error Occurred in Second Promise"), 2000));
        console.log(p2);  // Not executed due to the rejection & aborted

        let p3 = await new Promise((resolve) => setTimeout(() => resolve("Third
        Promise"), 500));
        console.log(p3);
  } catch (error) {
    console.error("Caught an error:", error);
  }
}
execute();
```

Example: Chaining Multiple await Statements in a Loop

If you need to handle multiple asynchronous operations in a loop, you can use await inside the loop to process each promise one by one.

```
async function execute() {
        let numbers = [1, 2, 3, 4, 5];

        for (let num of numbers) {
                let result = await new Promise((resolve) => setTimeout(() =>
                resolve(num * 2), 1000));
                console.log(result);
  }}
execute();
```

Example: Mixing Synchronous and Asynchronous Code

Sometimes, you may have both synchronous values and asynchronous operations, and you can handle them with multiple await statements.

```
async function execute() {
        let syncValue = 10;  // Synchronous value
        let asyncValue1 = await new Promise((resolve) => setTimeout(() =>
        resolve(syncValue * 2), 1000));
        console.log(asyncValue1);
        let asyncValue2 = await new Promise((resolve) => setTimeout(() =>
        resolve(asyncValue1 + 5), 2000));
        console.log(asyncValue2);
}
execute();
```

Example: Fetching User Data and Posts Sequentially, & Fetching Comments in Parallel

```
async function execute() {
        // Sequential part: Fetching user data and posts one after another
        let user = await new Promise((resolve) => {
          setTimeout(() => resolve({ id: 1, name: "Neha Rathore" }), 1000);
        });
          console.log("User:", user);
        let posts = await new Promise((resolve) => {
                setTimeout(() => resolve(["Post 1", "Post 2", "Post 3"]), 1000);
        });
         console.log("Posts:", posts);
        // Parallel part: Fetching comments for each post in parallel
        let comments = await Promise.all(posts.map(post =>
                    new Promise((resolve) => {
                      setTimeout(() => resolve(`Comments for ${post}`), 1000);
                    })
        ));
        console.log("Comments:", comments);
}
execute();
```

Propagating errors in async functions and Error Handling

Example: Propagating Errors with throw in async Functions

In the example, an error is thrown inside an async function and caught by the calling function using try/catch.

```
async function errorPropagation() {
        throw new Error("Error Occured!");  // Propagating error using throw
}

async function execute() {
  try {
        await errorPropagation();
  } catch (error) {
        console.error("Caught error:", error.message);
  }
}

execute();
```

Example: Propagating Errors in a Promise Chain

If an async function returns a rejected promise, the error can propagate to another async function that is awaiting it.

```
async function errorPropagation() {
        return Promise.reject("Promise Failed");  // Rejecting promise with an error
                                                  message
}
async function execute() {
  try {
        await errorPropagation();  // Error will propagate here
  } catch (error) {
        console.error("Caught error:", error);
  }
}
execute();
```

Example: Propagating Errors from Nested Async Functions

Errors can propagate through multiple async function calls.

```
async function nestedError() {
        throw new Error("Nested error occurred!");
}
async function errorPropagation() {
        await nestedError();  // Error propagates here from nested function
}
async function execute() {
try {
        await errorPropagation();  // Error propagates here from errorPropagation
  } catch (error) {
        console.error("Caught error:", error.message);
  }}
execute();
```

Example: Propagating Errors with Promise.all

When working with Promise.all, if one of the promises rejects, the entire Promise.all operation fails, and the error propagates.

```
async function errorPromise1() {
        return new Promise((resolve) => setTimeout(() => resolve("Promise 1"),
        1000));          }
async function errorPromise2() {
        return Promise.reject("Error in Promise 2!");  // Rejecting the promise
}
async function errorPropagation() {
  try {
        let results = await Promise.all([errorPromise1(), errorPromise2()]);
        console.log(results);  // Not get execute due to the error in Promise 2
  }
catch (error) {
        console.error("Caught error:", error);
  }}
errorPropagation();
```

Example: Propagating Errors Inside a Loop

When using await in loops, errors can be propagated from asynchronous operations inside the loop and caught in the calling function.

```
async function processItem(item) {
   if (item === 'cooler') {
        throw new Error(`Sending failed for item ${item}`);  // Throwing error for
                                                                item 'cooler'
   }
   return `${item} sent to customer`;
 }

 async function execute() {

 try {
        let item=['chair','table','cooler','fridge','fan']

        for (let i = 0; i <item.length; i++) {
                let result = await processItem(item[i]);  // Error propagates from
                                                             processItem

                console.log(result);  // Outputs until item 'cooler', then stops
                }
   }
catch (error) {
        console.error("Caught error:", error.message);
   }
 }

 execute();
```

Pitfalls of using await inside loops (sequential execution)

Using await inside loops can lead to sequential execution, which may not be optimal for performance, especially when multiple asynchronous operations can be executed concurrently.

- **Sequential Execution Slowing Down Processing:** When you use await in a loop, each iteration waits for the previous one to complete before proceeding, leading to slower execution time.

Example

```
function fetchData(value) {
        return new Promise(resolve => {
                setTimeout(() => {
                        console.log(`Fetched data for: ${value}`);
                        resolve(value);
                }, 1000); // Simulate network delay
        });
}

async function execute() {
        const values = [10, 20, 30, 40, 50];

        for (const value of values) {
                await fetchData(value);  // Awaits each fetch sequentially
        }
}

execute();  // Takes ~5 seconds to complete
```

points to note

- **Performance Impact**: In the example, fetching data for five values takes approximately 5 seconds because each fetchData call is awaited sequentially.
- **Better Approach**: Instead of awaiting inside the loop, you can start all operations simultaneously using Promise.all.

- **Resource Inefficiency:** If you have many iterations, using await in a loop can be resource-intensive and lead to inefficient use of resources.

Example

```
function fetchData(value) {
        return new Promise(resolve => {

                setTimeout(() => {
                        console.log(`Fetching: ${value}`);
                        resolve(value);
                        }, 1000); // Simulate network delay
        });
 }

async function fetchMultipleResources() {

        const resources = ['resource1', 'resource2', 'resource3', 'resource4'];

        for (const resource of resources) {
                const data = await fetchData(resource);  // Awaits each fetch
                console.log(`Fetched data for : ${data}`);
        }
}

fetchMultipleResources();
```

points to note

- **Inefficiency:** Each resource is processed one after the other, leading to unnecessary wait times. This is particularly inefficient if each operation can be independent of the others.
- **Optimization:** Again, using Promise.all to fetch all resources at once would be more efficient.

- **Losing the Benefits of Concurrency:** By awaiting inside a loop, you lose the benefits of concurrency, which can significantly slow down operations that can run in parallel.

Example

```
function fetchData(value) {
        return new Promise(resolve => {

                setTimeout(() => {
                        console.log(`Fetching: ${value}`);
                        resolve(value);
                        }, 1000); // Simulate network delay
        });
 }

async function concurrentExecution() {
        const results = [];
        const tasks = [10, 20, 30, 40, 50];

        for (const task of tasks) {
           const result = await fetchData(task);  // Sequential execution
           results.push(result);
         }

        console.log(results);
}

concurrentExecution();
```

points to note

- **Loss of Concurrency:** Each fetchData call blocks the next one until it's resolved. The total wait time becomes 5 seconds.
- **Improvement:** Using Promise.all allows all fetch operations to start simultaneously, leading to a reduction in total execution time.

- **Unintended Consequences with Large Data Sets:** Using await inside a loop can have unintended consequences when dealing with larger data sets, as it can lead to long execution times and potential throttling issues.

Example

```
function fetchData(value) {
        return new Promise(resolve => {

                setTimeout(() => {
                        console.log(`Fetching: ${value}`);
                        resolve(value);
                        }, 1000); // Simulate network delay
        });
 }

async function processLargeDataSet() {
        const largeDataSet = Array.from({ length: 100 }, (_, i) => i + 1);
        // Create an array from 1 to 100

        for (const data of largeDataSet) {
                await fetchData(data);
                // Sequentially processing a large data set
        }
}

processLargeDataSet();
```

points to note

- **Impact on Performance:** In this scenario, processing a large data set sequentially takes a significant amount of time (100 seconds).
- **Throttling Risk:** If these requests hit an API, you may also encounter throttling or rate-limiting issues due to the high number of sequential requests.
- **Alternative:** Using batching or running requests concurrently with Promise.all can simplify the performance concerns.