JavaScript Async-Await

Resolving JavaScript Promises

When using JavaScript `async...await`, multiple asynchronous operations can run concurrently. If the resolved value is required for each promise initiated, `Promise.all()` can be used to retrieve the resolved value, avoiding unnecessary blocking.

```
let promise1 = Promise.resolve(5);
let promise2 = 44;
let promise3 = new Promise(function(resolve, reject) {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([promise1, promise2,
promise3]).then(function(values) {
  console.log(values);
});
// expected output: Array [5, 44, "foo"]
```

Creating `async` Function

An asynchronous JavaScript function can be created with the `async` keyword before the `function` name, or before `()` when using the arrow function syntax. An `async` function returns a promise.

```
function helloWorld() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('Hello World!');
    }, 2000);
  });
}

const msg = async function() { //Async Function Expression
  const msg = await helloWorld();
  console.log('Message:', msg);
}

const msg1 = async () => { //Async Arrow Function
  const msg = await helloWorld();
  console.log('Message:', msg);
}

msg(); // Message: Hello World! <-- after 2 seconds
msg1(); // Message: Hello World! <-- after 2 seconds
```

Async Await Promises

The `async...await` syntax in ES6 offers a new way write more readable and scalable code to handle promises. It uses the same features that were already built into JavaScript.

```
function helloWorld() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('Hello World!');
    }, 2000);
  });
}

async function msg() {
  const msg = await helloWorld();
  console.log('Message:', msg);
}

msg(); // Message: Hello World! <-- after 2 seconds
```

Using async await syntax

Constructing one or more promises or calls without `await` can allow multiple `async` functions to execute simultaneously. Through this approach, a program can take advantage of *concurrency*, and asynchronous actions can be initiated within an `async` function. Since using the `await` keyword halts the execution of an `async` function, each async function can be awaited once its value is required by program logic.

JavaScript async...await advantage

The JavaScript `async...await` syntax allows multiple promises to be initiated and then resolved for values when required during execution of the program. As an alternate to chaining `.then()` functions, it offers better maintainablity of the code and a close resemblance synchronous code.

Async Function Error Handling

JavaScript `async` functions uses `try...catch` statements for error handling. This method allows shared error handling for synchronous and asynchronous code.

```
let json = '{ "age": 30 }'; // incomplete data

try {
  let user = JSON.parse(json); // <-- no errors
  alert( user.name ); // no name!
} catch (e) {
  alert( "Invalid JSON data!" );
}
```

codecademy

The `aysnc` and `await` Keywords

The `async ... await` ES6 JavaScript syntax offers a new way to write more readable and scalable code to handle promises. A JavaScript `async` function can contain statements preceded by an `await` operator. The operand of `await` is a promise. At an `await` expression, the execution of the `async` function is paused and waits for the operand promise to resolve. The `await` operator returns the promise's resolved value. An `await` operand can only be used inside an `async` function.

```
function helloWorld() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('Hello World!');
    }, 2000);
  });
}

async function msg() {
  const msg = await helloWorld();
  console.log('Message:', msg);
}

msg(); // Message: Hello World! <-- after 2 seconds
```

Print