codecademy

Apprendre la syntaxe JavaScript : Async-Await

Résoudre les promesses JavaScript

Lorsque vous utilisez JavaScript `async...await`, plusieurs opérations asynchrones peuvent s'exécuter simultanément. Si la valeur résolue est requise pour chaque promesse initiée, `Promise.all()` peut être utilisée pour récupérer la valeur résolue, en évitant un blocage inutile.

```
soit promesse1 = Promesse . résoudre ( 5 ) ;
soit promesse2 = 44 ;
laissez promesse3 = nouvelle promesse ( fonction ( résoudre , rejeter ) {
  setTimeout ( résoudre , 100 , 'foo' ) ;
} ) ;

Promesse . tout ( [ promesse1 , promesse2 , promesse3 ] ) . alors ( fonction ( valeurs ) {
  consoler . log ( valeurs ) ;
} ) ;
// sortie attendue : Array [5, 44, "foo"]
```

Créer asyncune fonction

Une fonction JavaScript asynchrone peut être créée avec le `async` mot-clé avant le `function` nom, ou avant `()` lors de l'utilisation de la syntaxe de la fonction fléchée. Une `async` fonction renvoie une promesse.

```
fonction bonjourMonde ( ) {
  retourner une nouvelle promesse ( résoudre => {
    setTimeout ( ( ) => {
      résoudre ( 'Bonjour le monde !' ) ;
    } , 2000 ) ;
  } ) ;
}

const msg = fonction asynchrone ( ) { // Expression de fonction asynchrone
  const msg = attendre helloWorld ( ) ;
  consoler . 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!" );
}
```

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
```