Theory: Async/await

26 minutes 0 / 5 problems solved

In the previous topics, we've learned how to create a *Promise* and how to use the methods `.then`, `.catch,` and `.finally`. to handle Promise results. In this topic, we will learn a simpler way to work with Promise with a more legible code.

The keywords "async/await" allow us to create asynchronous functions that will always return a *Promise*, without having to explicitly create a *new Promise* and ending up with *Promises chaining*.

§1. Async

Let's start with the `async` keyword.

We use it before a function declaration, like this: `async function`. This way we define that our function will be an asynchronous function.

Let's observe a simple example:

```
async function foo() {
  return 'bar';
}

console.log(foo()); // Promise { 'bar' }
```

As you can see, it is unnecessary to explicitly return a *Promis*e because the return of an async function will always be a *Promise*.

To get a better understanding of how this works let's create two functions:

- searchEngine(engine): This function will receive one *argument* and will return a *Promise* if the parameter is equal to `"Google"`. For now, the function will return only a resolved Promise.
- handlePromiseResult(): This function will save the result of the `searchEngine()` function to the `resultOfThePromise` constant.

The result will be the following:

```
async function searchEngine(engine) {
  if (engine === 'Google') {
    return Promise.resolve('You can start googling!');
  }
}
```

```
function handlePromiseResult() {
  const resultOfThePromise = searchEngine('Google');

  resultOfThePromise
    .then(response => console.log(response)); // You can start googling!
}

handlePromiseResult();
```

The `searchEngine()` function returned a Promise as expected, otherwise we wouldn't be able to use the `.then()` method. Then we stored the result in the constant `resultOfThePromise` and used the `.then()` method and displayed the response in the console.

§2. Await

You can see the principal advantage of using `async` function when you combine it with `await`.

Current topic:

Async/await

Topic depends on:

× Arrow functions

× "then", "catch" and "finally" methods

Table of contents:

Instead of using `promise.then()` we can use this syntax to handle Promise result. We use the keyword `await` always within an asynchronous function and place it before the return of an asynchronous function, which is easier.

```
async function handlePromiseResult() {

const resultOfThePromise = await searchEngine('Google'); // Wait until the promise resolves.
}
```

Continuing with our `searchEngine()` function what will happen if our Promise takes some time to fulfill and return the result? We will change the `searchEngine()` function to return a Promise after 2 seconds.

```
async function searchEngine(engine) {
  return new Promise((resolve, reject) => {
    if (engine === 'Google') {
      setTimeout(() => resolve('You can start googling!'), 2000);
    }
  });
}
```

Now we need to make the function `handlePromiseResult()` wait for our Promise to be fulfilled.
That's how it should be done:

```
async function handlePromiseResult() {

const resultOfThePromise = await searchEngine('Google'); // Wait until the promise resolves.

  console.log(resultOfThePromise); // You can start googling!
}

handlePromiseResult();
```

When we call the function `searchEngine('Google')`, `await` suspends the execution of the function and waits until the Promise is fulfilled. When the Promise is fulfilled, the await resumes the function, stores the result in the constant `resultOfThePromise` and then displays the result in the console.

If we use `await` outside of an asynchronous function, we will receive a SyntaxError

```
function handlePromiseResult() {
  const resultOfThePromise = await searchEngine('Google');


console.log(resultOfThePromise); // SyntaxError: await is only valid in async function
}

handlePromiseResult();
```

Our `handlePromiseResult()` function returned an error: SyntaxError: await is only valid in async function.
It happened because `await` can't be used in non-async function.

§3. Exception Handling

Our `searchEngine()` function is still incomplete. We should improve it so that the Promise will be rejected if the parameter is different from `"Google"`.

```
async function searchEngine(engine) {
  return new Promise((resolve, reject) => {
    if (engine === 'Google') {
      setTimeout(() => resolve('You can start googling!'), 2000);
    } else {
      reject('Sorry! Only Google is allowed.');
    }
  });
}
```

When the Promise is fulfilled, `await` returns the expected result, but if it's rejected, it throws an error. We can catch the error using `try..catch` statements.

Now if we call our `handlePromiseResult()` function, we will get the `Unhandled promise rejection` error . That's because we need to handle ***reject*** case. However, rather than implementing `.catch()` method we will implement the `try..catch` statements.

```
async function handlePromiseResult() {
  try {
    const resultOfThePromise = await searchEngine('Bing');
    console.log(resultOfThePromise);
  } catch(err) {
    console.log(err); // Sorry! Only Google is allowed.
  }
}


handlePromiseResult();
```

Here you can see that instead of using the `.then()` and `.catch()` methods, we used the `try..catch` keywords. Here's how they work:

- First everything between `try` and `catch` will be executed.
- In case the Promise is resolved, the `try` block is executed and finished successfully.
- In case the Promise is rejected, `catch` captures the error and the block is executed.

§4. Conclusion

We've learned two keywords to work with Promises: `async` and `await`.

The `async` keyword has two main applications. It makes a function always return a Promise and allows us to use `await` to pause a function and resume it when the Promise is fulfilled.

The `await` keyword makes a function wait until a Promise is fulfilled. Like the ordinary `.then()` method, `await` makes the function wait for the Promise to be fulfilled. It always has to be used within an asynchronous function.

We've also come across the `try..catch` statements to handle rejections and errors. Now let's get down to the code challenges!

Report a typo

17 users liked this theory. **1** didn't like it. **What about you?**

Start practicing

Comments (0) Hints (0) Useful links (1) Show discussion