Tip 44: Create Clean Functions with Async/Await

In this tip, you'll learn how to streamline promises with async/await.

We'll cover the following

In the previous tip, you saw that promises are awesome. They're a vast improvement over callbacks, but their interfaces are still a little clunky. You're still working with callbacks in methods. Fortunately, the language continues to improve. You can now avoid callbacks entirely by adding *asynchronous promise* data to a variable in a single function.

`async` & `await`

Developers usually talk about the *new* syntax, `async`/`await`, as a group, but it's really two *separate* actions. You use the `async` keyword to *declare* that an encapsulating *function* will be using *asynchronous* data. Inside the *asynchronous* function, you can use the `await` keyword to *pause* the function until a value is *returned*.

Before you begin, there are a couple things to note.

- First, this *doesn't* replace promises. You're merely wrapping promises in a better syntax.
- Second, it isn't well supported, and the compiled code is a little buggy. It's safe to use on server-side JavaScript, but you may have problems in browsers.

Example

To see `async`/`await` in action, refactor some of your code from the previous tip. As a reminder, you pass a function to the `then()` method on the `getUserPreferences()`

function.

```
function getUserPreferences() {
    const preferences = new Promise((resolve, reject) => {
        resolve({
            theme: 'dusk',
        });
    });
    return preferences;
}

getUserPreferences()
    .then(preferences => {
        console.log(preferences.theme);
    });
```

First, you'll need to wrap the call to `getUserPreferences()` in another function. Write a *new* function called `getTheme()`. This will hold all of your calls to asynchronous functions. To indicate that you'll be calling asynchronous functions, add the `async` keyword right before the `function` keyword.

Inside your `getTheme()` function, you can call `getUserPreferences()`. Before you call the function, though, add the `await` keyword to signal that `getUserPreferences()` will return a *promise*. This allows you to assign the *resolved* promise to a *new* variable.

```
async function getTheme() {
    const { theme } = await getUserPreferences();
    return theme;
}
console.log(getTheme());
```

The trick with an asynchronous function is that it's transformed into a promise. In other words, when you call `getTheme()`, you'll still need a `then()` method.

```
async function getTheme() {
    const { theme } = await getUserPreferences();
    return theme;
}

getTheme()
    .then(theme => {
        console.log(theme);
    });
```

You cleaned things up a little, but honestly, not much. `async` functions really shine when you're working with multiple promises.

Using `async` with multiple promises

Think about the previous tip when you chained *multiple* promises together. With `async`/`await`, you can assign each *return* statement to a *variable* before passing the variable to the *next* function. In other words, you can transform your chained promises into a series of function calls in a single wrapping function. Try creating a new function called `getArtistsByPreference()` where you call a *series* of asynchronous functions passing the data from the previous function as an argument to the next.

```
async function getArtistByPreference() {
    const { theme } = await getUserPreferences();
    const { album } = await getMusic(theme);
    const { artist } = await getArtist(album);
    return artist;
}
getArtistByPreference()
    .then(artist => {
        console.log(artist);
    });
```

That's a vast improvement over a long method chain.

Handling errors

All that's left is handling *errors*. In this case, you'll need to move error handling outside the wrapping function. Instead, you still use the `catch` method when you're invoking `getArtistsByPreference()`. Because `getArtistsByPreference()` returns a promise, you need to add a `catch()` method in case any of your internal asynchronous functions return an *error*.

```
async function getArtistByPreference() {
    const { theme } = await getUserPreferences();
    const { album } = await failMusic(theme);
    const { artist } = await getArtist(album);
    return artist;
}

getArtistByPreference()
    .then(artist => {
        console.log(artist);
    })
    .catch(e => {
        console.error(e);
    });
```

`async` / `await` functions can clean up your code, but again, use them with caution, particularly when you're compiling your code to earlier versions of JavaScript.

At this point, you have a few tools for handling asynchronous actions, but you're probably wondering when you'll actually use them. You use promises in many situations, but the most common is when you're fetching data from an API.

Q

What will be the output of the code below?

```
const add = function(x,y){
  return new Promise((resolve,reject) => {
    resolve(x+y);
  })
}

const multiply = function(x,y){
  return new Promise((resolve,reject) => {
    resolve(x*y);
  })
}

async function f() {
  let result1 = await add(2,5);
  let result2 = await multiply(9,4);
  return result1 + result2;
}

f()
.then((data) => console.log(data))
.catch((data) => console.log(data))
```

Retake Quiz

In the next tip, you’ll learn how to access data from an endpoint using fetch.