Tip 43: Retrieve Data Asynchronously with Promises

In this tip, you'll learn how to work with delayed data responses using promises.

We'll cover the following

What is an asynchronous language?

JavaScript is an *asynchronous* language. That's a big word for a simple concept. An asynchronous language is merely a language that can execute subsequent lines of code when previous lines of code aren't fully resolved.

All right. Maybe that explanation wasn't any more clear. Think about reasons why code may be blocked. You may be getting data from an API. You might be pulling data from the DOM or other source. You might be waiting for a user response. The common thread is you need some information to proceed, and it may take time to get it. If you want more examples, Peter Olson has a great breakdown of the differences between asynchronous and synchronous code.

The value of asynchronous languages is that if there are parts of your code that don't require the delayed information, you can run the code while the other code is waiting. If you're waiting for an API response, you can still respond to click methods on other elements or calculate values of other data sources. Your code

doesn't grind to a halt while waiting.

In later tips, you'll work with API data specifically. In this tip, you'll explore a reusable technique for working with asynchronous data: *promises*.

Using callbacks for asynchronous actions

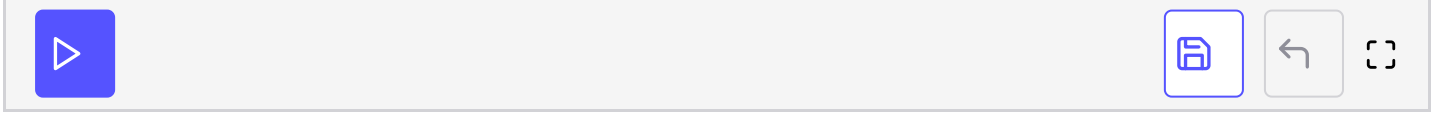
Before promises, developers used *callbacks* to handle asynchronous actions. If you requested expenses from a data source, you'd pass a callback function as an argument. After the asynchronous data is *returned—or resolved* as it is often called —the function would *execute* the callback. The traditional example is a `setTimeout()` function that takes a callback and executes it after a certain number of *milliseconds*.

Use `setTimeout()` as a place holder for any action that doesn't immediately resolve. For example, think about a function called `getUserPreferences()`, which would fetch data from an API and then pass that data to a callback.

Because Javascript is asynchronous, you can call other functions before and after the call to `getUserPreferences()` and they'd both resolve before `getUserPreferences()` executes the callback.

Example: Using `setTimeout()`

```
function getUserPreferences(cb) {
    setTimeout(() => {
        cb({
            theme: 'dusk',
        });
    }, 1000);
}
function log(value) {
    return console.log(value);
}
log('starting');
getUserPreferences(preferences => {
    return log(preferences.theme.toUpperCase());
});
log('ending?');
```



Nested callbacks

Callbacks are a fine way to handle asynchronous data. And they were a standard

tool for a long time. The problem is that you may have asynchronous functions that call asynchronous functions, that call asynchronous... Eventually you have so many nested callbacks, you find yourself in what became known as **"callback hell"**.

What if you wanted to get a music selection based on a user's preference? The callback function, `getMusic()`, also needs to hit an API and also needs a callback based on the API response. You encounter this situation all the time. Here's your `getMusic()` function:

```
function getMusic(theme, cb) {
    setTimeout(() => {
        if (theme === 'dusk') {
            return cb({
                album: 'music for airports',
            });
        }
        return cb({
            album: 'kind of blue',
        });
    }, 1000);
}
```

Now you need to get the preference and then get an album. First, you'd make a call to `getUserPreferences()` and you'd pass `getMusic()` as a callback. `getMusic()` takes a theme preference and a callback. This function is only nested two deep, and it's already getting hard to read.

```
function getMusic(theme, cb) {
    setTimeout(() => {
        if (theme === 'dusk') {
            return cb({
                album: 'music for airports',
            });
        }
        return cb({
            album: 'kind of blue',
        });
    }, 1000);
}

getUserPreferences(preferences => {
    return getMusic(preferences.theme, music => {
        console.log(music.album);
    });
});
```

As if that weren't enough, many asynchronous functions took *two* callbacks: a callback for a *successful* response and a callback for an *error*. Things got complicated fast.

Promises

Promises solve the callback problem twice over. Instead of taking callback functions as arguments, promises have *methods* for success and failure. This keeps things visually flat. In addition, you can *chain* together asynchronous promises instead of nesting them. This means that you can neatly stack all of your actions.

How do promises work?

A **promise** is an object that takes *asynchronous* action and calls *one* of two methods based on the response. If the asynchronous action is *successful*, or *fulfilled*, the promise passes the results to a `then()` method. If the action *fails*, or is *rejected*, the promise calls the `catch()` method. Both `then()` and `catch()` take a function as an argument, and that can only take a single response argument.

A promise takes two arguments: `resolve()` and `reject()`. `resolve()` is what happens when things go as planned. When `resolve()` is called, the code will execute the function passed to the `then()` method. When you define your `getUserPreferences()` function, you'll return the promise. When you actually call `getUserPreferences()`, you'll call either the `then()` or the `catch()` method.

Example

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

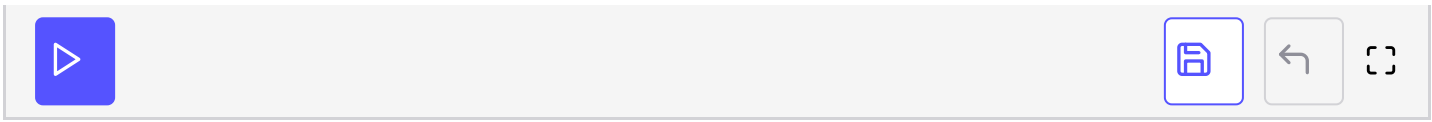
Here's an example of calling a code and running a function on successful resolution using the `then()` method.

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

```
    return preferences;
}



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



`then()` & `catch()`

In the above case, things went well, but you should always have a backup plan. Whenever you set up a promise, you can have both a `then()` and a `catch()` method. The `then()` method will handle the *resolutions*. The `catch()` method will handle the *rejections*.

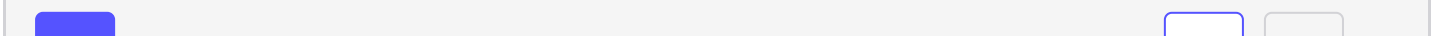
Here's a failing promise. Note that it's calling the `reject()` argument.

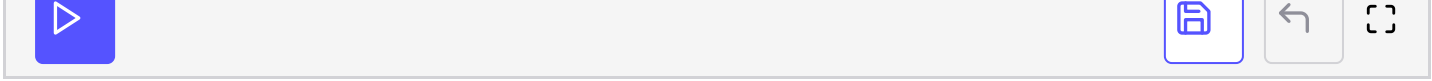
```
function failUserPreference() {
    const finder = new Promise((resolve, reject) => {
        reject({
            type: 'Access Denied',
        });
    });
    return finder;
}
```

When you call a promise, you can add attach the `then()` method and the `catch()` using chaining.

```
function failUserPreference() {
    const finder = new Promise((resolve, reject) => {
        reject({
            type: 'Access Denied',
        });
    });
    return finder;
}

failUserPreference()
    .then(preferences => {
        // This won't execute
        console.log(preferences.theme);
    })
    .catch(error => {
        console.error(`Fail: ${error.type}`);
    });
// Fail: Access Denied
```





This code should already look more clean. The fun really begins when you chain multiple promises together.

Chaining promises

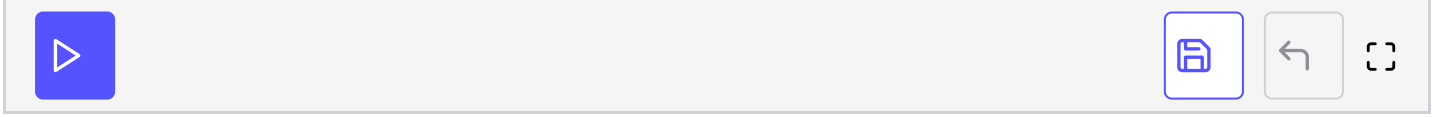
Remember your `getMusic()` function? Try converting that to a *promise*.

```
function getMusic(theme) {
    if (theme === 'dusk') {
        return Promise.resolve({
            album: 'music for airports',
        });
    }
    return Promise.resolve({
        album: 'kind of blue',
    });
}
```

After you do, you can call and return it in the `then()` method of `getUserPreferences()`. After you do that, you can call another `then()` method, which will call a function using the results from `getMusic()`.

```
function getMusic(theme) {
    if (theme === 'dusk') {
        return Promise.resolve({
            album: 'music for airports',
        });
    }
    return Promise.resolve({
        album: 'kind of blue',
    });
}

getUserPreferences()
    .then(preference => {
        return getMusic(preference.theme);
    })
    .then(music => {
        console.log(music.album);
    });
```



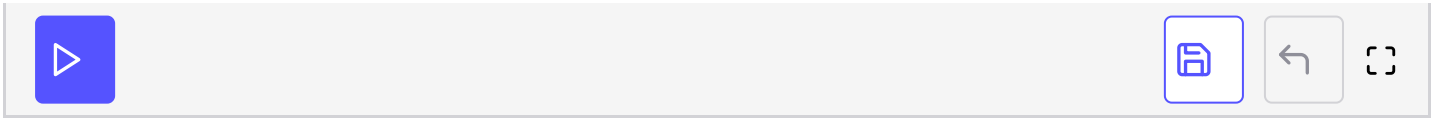
See what's happening? Instead of passing data into a series of nested callbacks, you're passing data down through a series of `then()` methods.

And, of course, because you're returning promises, you can convert everything to *single-line arrow functions with an implicit return*.

single-line arrow functions with an implicit return.

```
function getMusic(theme) {
    if (theme === 'dusk') {
        return Promise.resolve({
            album: 'music for airports',
        });
    }
    return Promise.resolve({
        album: 'kind of blue',
    });
}

getUserPreferences()
    .then(preference => getMusic(preference.theme))
    .then(music => { console.log(music.album); });
```



Finally, as if that weren't enough, if you're chaining promises together, you don't need to add a `catch()` method to each one. You can define a *single* `catch()` method to handle a case where any promise is *rejected*.

Chained `catch()` statement

To see a chained `catch()` in action, create another promise that returns the `artist` for an album.

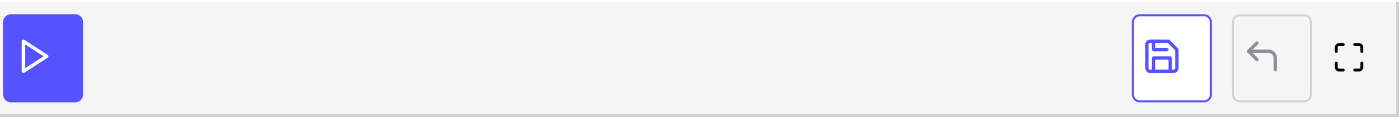
```
function getArtist(album) {
    return Promise.resolve({
        artist: 'Brian Eno',
    });
}
```

Unfortunately, you won't get to use `getArtists()` because `getMusic()` is going to be rejected. Don't worry—it won't kill your code. Your code will execute the `catch()` at the bottom of the group even though it was defined after another `then()` method.

```
function getArtist(album) {
    return Promise.resolve({
        artist: 'Brian Eno',
    });
}

function failMusic(theme) {
    return Promise.reject({
        type: 'Network error',
    });
```

```
}
getUserPreferences()
    .then(preference => failMusic(preference.theme))
    .then(music => getArtist(music.album))
    .catch(e => {
        console.log(e);
    });
```



As you can see, promises can handle a lot of situations with a very simple interface. There's even a method called Promise.all that takes an *array* of promises and returns either a *resolve* or a *reject* when they all finish.

Promises took the JavaScript world by storm. They're an amazing tool that can help you make otherwise ugly code gorgeous and easy to read.

Of course, things can always get better. In ES2017, the TC39 committee approved a new method for handling asynchronous functions. Well, it's actually a *two-step* process called `async` / `await` and it takes asynchronous data in an interesting new direction.

1 `setTimeout()` runs code asynchronously in JavaScript. True or false?

2 What will be the output of the code below?

```
var multiply = function(num1, num2) {
  return new Promise((resolve,reject) => {
    var product = num1 * num2;
    if (product) {
      resolve(product);
    }
    else {
      reject(Error("Could not multiply the values!"));
```

```
    }
  });
};


multiply(2,"x")
  .then((data) => {console.log(data)})
  .catch((error) => {console.log(error)})
```

3

What will be the output of the code below?

```
var multiply = function(num1, num2) {
  return new Promise((resolve,reject) => {
    var product = num1 * num2;
    if (product) {
      resolve(product);
    }
    else {
      reject(Error("Could not multiply the values!"));
    }
  });
};

var double = function(val){
  return Promise.resolve(val*2)
}

multiply(2,2)
  .then((data) => double(data))
  .then((data) => console.log(data))
  .catch((error) => {console.log(error)})
```

Retake Quiz

In the next tip, you'll explore `async`/`await` and see how you can make your

In the next tip, you'll explore `async`/`await` and see how you can make your asynchronous code even more readable.