October 18, 2022

Promisification

“Promisification” is a long word for a simple transformation. It’s the conversion of a function that accepts a callback into a function that returns a promise.

Such transformations are often required in real-life, as many functions and libraries are callback-based. But promises are more convenient, so it makes sense to promisify them.

For better understanding, let’s see an example.

For instance, we have `loadScript(src, callback)` from the chapter Introduction: callbacks.

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Script load error for ${src

  document.head.append(script);
}

// usage:
// loadScript('path/script.js', (err, script) => {...})
```

The function loads a script with the given `src`, and then calls `callback(err)` in case of an error, or `callback(null, script)` in case of successful loading. That’s a widespread agreement for using callbacks, we saw it before.

Let’s promisify it.

We’ll make a new function `loadScriptPromise(src)`, that does the same (loads the script), but returns a promise instead of using callbacks.

In other words, we pass it only `src` (no `callback`) and get a promise in return, that resolves with `script` when the load is successful, and rejects with the error otherwise.

Here it is:

```
let loadScriptPromise = function(src) {
  return new Promise((resolve, reject) => {
    loadScript(src, (err, script) => {
      if (err) reject(err);
      else resolve(script);
    });
  });
};

// usage:
// loadScriptPromise('path/script.js').then(...)
```

As we can see, the new function is a wrapper around the original `loadScript` function. It calls it providing its own callback that translates to promise `resolve/reject`.

Now `loadScriptPromise` fits well in promise-based code. If we like promises more than callbacks (and soon we'll see more reasons for that), then we will use it instead.

In practice we may need to promisify more than one function, so it makes sense to use a helper.

We'll call it `promisify(f)`: it accepts a to-promisify function `f` and returns a wrapper function.

```
function promisify(f) {
  return function (...args) { // return a wrapper-function (*)
    return new Promise((resolve, reject) => {
      function callback(err, result) { // our custom callback for f (**)
        if (err) {
          reject(err);
        } else {
          resolve(result);
        }
      }

      args.push(callback); // append our custom callback to the end of t

      f.call(this, ...args); // call the original function
    });
  };
}

// usage:
let loadScriptPromise = promisify(loadScript);
loadScriptPromise(...).then(...);
```

The code may look a bit complex, but it's essentially the same that we wrote above, while promisifying `loadScript` function.

A call to `promisify(f)` returns a wrapper around `f` `(*)`. That wrapper returns a promise and forwards the call to the original `f`, tracking the result in the custom callback `(**)`.

Here, `promisify` assumes that the original function expects a callback with exactly two arguments `(err, result)`. That's what we encounter most often. Then our custom callback is in exactly the right format, and `promisify` works great for such a case.

But what if the original `f` expects a callback with more arguments `callback(err, res1, res2, ...)`?

We can improve our helper. Let's make a more advanced version of `promisify`.

- When called as `promisify(f)` it should work similar to the version above.
- When called as `promisify(f, true)`, it should return the promise that resolves with the array of callback results. That's exactly for callbacks with many arguments.

```
// promisify(f, true) to get array of results
function promisify(f, manyArgs = false) {
  return function (...args) {
    return new Promise((resolve, reject) => {
      function callback(err, ...results) { // our custom callback for f
        if (err) {
          reject(err);
        } else {
          // resolve with all callback results if manyArgs is specified
          resolve(manyArgs ? results : results[0]);
        }
      }

      args.push(callback);

      f.call(this, ...args);
    });
  };
}

// usage:
f = promisify(f, true);
f(...).then(arrayOfResults => ..., err => ...);
```

As you can see it's essentially the same as above, but `resolve` is called with only one or all arguments depending on whether `manyArgs` is truthy.

For more exotic callback formats, like those without `err` at all: `callback(result)`, we can promisify such functions manually without using the helper.

There are also modules with a bit more flexible promisification functions, e.g. es6-promisify. In Node.js, there's a built-in `util.promisify` function for that.

Please note:

Promisification is a great approach, especially when you use `async/await` (covered later in the chapter Async/await), but not a total replacement for callbacks.

Remember, a promise may have only one result, but a callback may technically be called many times.

So promisification is only meant for functions that call the callback once. Further calls will be ignored.