Error Handling with Promises

Learn how ES2015 solves the problem of callback hell and the Pyramid of Doom. Promises are now used everywhere in asynchronous code and are part of modern and future JavaScript.

In this lesson, we'll learn how to handle errors with Promises. Here's the code we were using in the last lesson.

```
onRequest((request, response) =>
    readFile(request.fileToRead, data =>
        writeFile(request.fileToWrite, data, status =>
            response.send(status)
        )
    )
);
```

Let's add error handling to this function. We'll use a simple `try`/`catch` block to catch any possible errors.

```
onRequest((request, response) => {
    try {
        readFile(request.fileToRead, data =>
            writeFile(request.fileToWrite, data, status =>
                response.send(status)
            )
        );
    } catch(err) {
        console.log(err);
        response.send(err);
    }
});
```

Here's the Promise-enabled code we wrote last lesson.

```
onRequest((request, response) =>
    readFile(request.fileToRead)
    .then(data => writeFile(request.fileToWrite, data))


    .then(status => response.send(status))
);
```

Error Handling

Asynchronous operations generally require error handling in case something goes wrong (network issues, data not found, bad request, etc.). Promises have this built-in. They can take in a second callback that handles errors. It'll only be called in the case of an error.

Working Fine

Here, we here present the case where everything works correctly, but we've added error handling.

```
onRequest((request, response) =>
    readFile(request.fileToRead)
    .then(
        data => writeFile(request.fileToWrite, data),
        error => {
          console.log('Error when reading file:', error);
          return 'Failed';
        })
    .then(
        status => response.send(status),
        error => {
            console.log('Error when writing to file:', error);
            response.send(error);
        }
    )
);
```

This looks a lot hairier than the previous version, but we can figure out what's going on. All we've done is given a second callback to each of the two `.then` calls. These will run instead of the first callback if an error is thrown in the previous step.

If this still looks too hairy, we'll have a cleaner error handling solution towards the end of the lesson.

towards the end of the lesson.

Error Thrown

Let’s show the case where something goes wrong - let’s say that the file we’re trying to write to wasn’t found. The error starts on line 4. Run the code to see the output.

```
onRequest((request, response) =>
    readFile(request.fileToRead)
    .then(
        data => writeFile(request.fileToWrite, data),
        error => {
          console.log('Error when reading file:', error);
          return 'Failed';
        })
    .then(
        status => response.send(status),
        error => {
            console.log('Error when writing to file:', error);
            response.send(error);
        }
    )
);
```

An error is thrown during the `writeFile` process on line 4. This error cascades down to the next `.then`'s error handler on line 11, skipping the callback on line 10.

Error Handling when Promisifying

The callback to the promise constructor also takes in a second error-handling parameter. It’s commonly referred to as the `reject` parameter. If the operation was successful, we want to call `resolve`. Otherwise, call `reject`.

```
onRequest((request, response) => {
    const readFilePromise = new Promise((resolve, reject) => {
        readFile(request.fileToRead, (data, err) => {
            if (err) {
                reject(err);
                return;
            }

            resolve(data);
        });
    });

    readFilePromise
     .then(
        data => writeFile(request.fileToWrite, data),
        error => {
          console.log('Error when reading file:', error);
          return 'Failed';
        })
    .then(
        status => response.send(status),
        error => {
            console.log('Error when writing to file:', error);
            response.send(error);
        }
    );
});
```

If there's an error thrown during the `readFile` operation, it's now dealt with.

`.catch`

We have a cleaner way to deal with errors than passing in a second callback. We can attach a `.catch` method to the end of a Promise chain. The callback given to `.catch` will be called if *any* of the asynchronous functions results in an error.

This is useful when we don't particularly care about the errors thrown by any one of the asynchronous `.then` calls. If we only want to ensure that the entire Promise chain worked as expected, `.catch` is a fine solution.

Here, the `readFile` operation on line 2 throws an error.

```
onRequest((request, response) =>
    readFile(request.fileToRead)
    .then(data => writeFile(request.fileToWrite, data))
    .then(status => response.send(status))
    .catch(error => {
        console.log('Caught error:', error);
        response.send(error);
    })
);
```

An error is thrown on line 2. Neither of the `.then`s on lines 3 or 4 are called. The engine goes immediately to the `.catch` method.

These aren’t going to be intuitive until you start using them in asynchronous code. In fact, they are often extremely hard to follow and get a grasp of conceptually. This should hopefully be enough to get you started.

That’s it for Promise error handling.

We’ll cover yet another asynchronous mechanism, `async`/`await`, in the next lesson.