Open in app Get started

Lucy Mitchell Follow
Jun 11, 2019 · 5 min read · Listen

Save

Using .then(), .catch(), .finally() to Handle Errors in Javascript Promises

What is error handling?

Building software that does what you want is great. Building robust software which pre-empts potential issues and can recover enough to give you (as a developer or a user) feedback is even better.

Error handling in programming refers to how errors are anticipated, detected, and resolved. It is a language-agnostic concept, and a positive way to approach your code; there's always something that could break, which means there's always something to potentially improve. For developers, the key outcome is evolution from "code hits bug, code crashes" to "code hits bug, code finds an alternative route to take, code does not crash but gives user feedback, because that's what you programmed it to do". When used appropriately, it's a graceful way of handling problems.

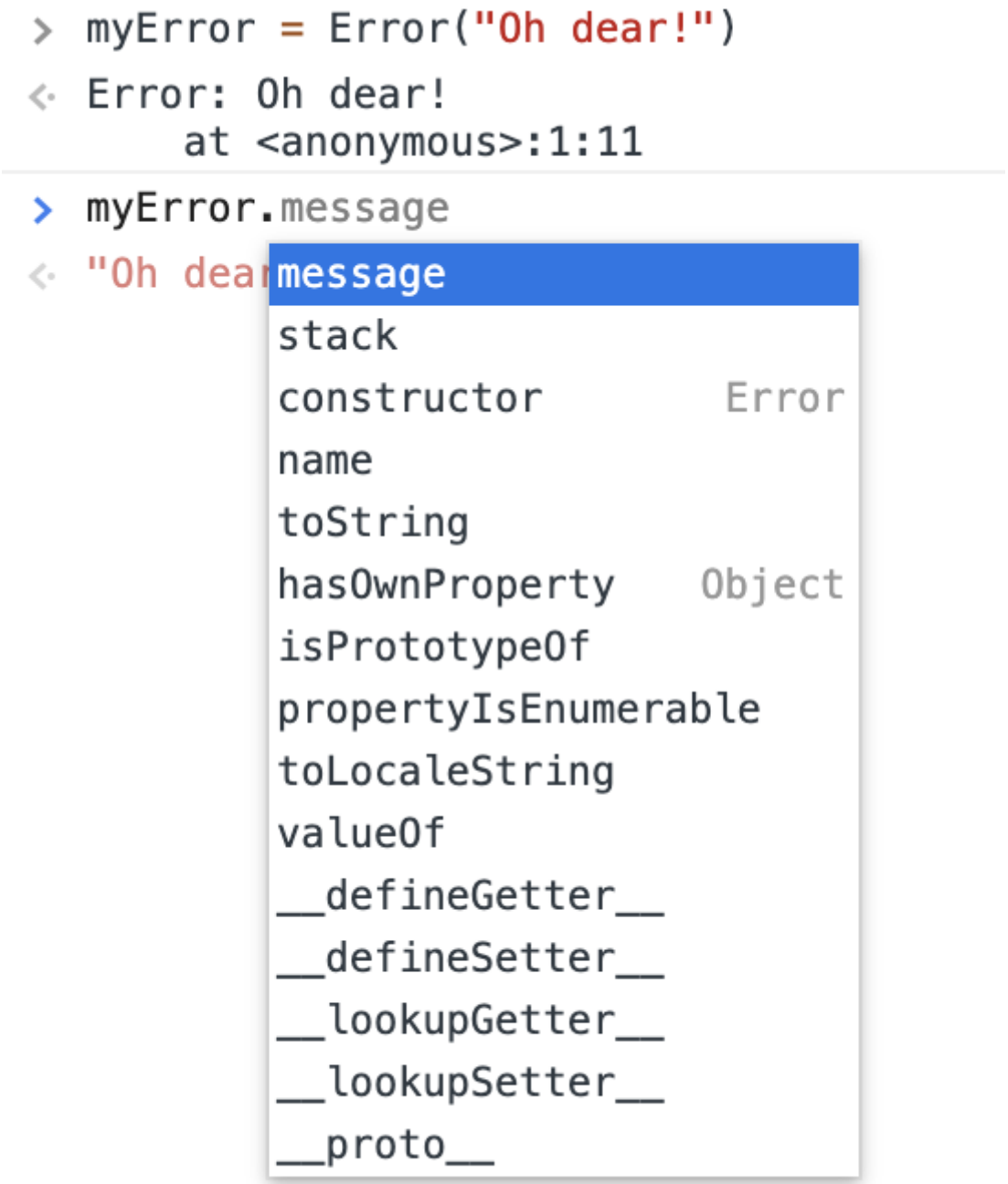
What does it look like in Javascript?

There are different ways to build error handling into your code, including the neat, built-in "try… catch" paradigm. The basic syntax looks like this:

```
try {
 // code that we will 'try' to run
} catch(error) {
 // code to run if there are any problems
}
```


Open in app Get started

For all built-in or generic errors, the error object inside the `catch` block comes with a handful of properties and methods:



[IMAGE DESCRIPTION: making a new error in the browser and seeing what comes with it; errors don't actually have any methods of their own, but do inherit some. Interestingly, you can make a new error either using the keyword "new" or omit it and call Error as a function, as I did above — it has the same functionality. You can create an Error instance with no parameters as all are optional and will revert to defaults, but I passed in a string message.]

Two particularly useful and most widely supported are:

Open in app Get started

Note: Javascript works by reading the code first, then running it. If your code is invalid, for example being syntactically incorrect from having unmatched curly braces somewhere, the Javascript engine won't being able to read it. This is a "parsetime error" as the code cannot be parsed properly. `try... catch` can only catch errors which occur at runtime (after this first reading/parsing phase) so your code *must* be valid in order for the `try... catch` to work.

try… catch… finally

A discerning addendum to this paradigm is the concept of `finally`. It kinda does what it says on the tin. To iterate on our previous example,

```
try {
  // code that we will 'try' to run
} catch(error) {
  // code to run if there are any problems
} finally {
  // run this code no matter what the previous outcomes
}
```

I really like Codeburst's example and encourage you to copy & paste it and try it out in the browser to see it in action. They also cover `throw` which is how to generate your own custom exceptions:

```
try {
   let hello = prompt("Type hello")
   if (hello !== 'hello'){
     throw new Error("Oops, you didn't type hello")
   }
} catch(e) {
   alert(e.message)
} finally {
   alert('thanks for playing!')
}
```

Caveat: you can nest `try... catch` statements! `catch` and `finally` clauses are, in theory both optional, though you need at least one of them. However if you don't


Open in app Get started

Second caveat: `try... catch` is a great approach, but shouldn't be used for everything ever. For a start, it can quickly make your code look verbose and repetitive. You also want to make sure you get into the habit of effectively handling errors, not just logging them out with this pattern.

`Try... catch` with Promises

Quick recap: in Javascript, a Promise is an object used as a proxy for a value not yet known.

It's called a Promise because it's saying "I may not know what the value of the return is right now, but I *promise* to return it at some point, stored inside this thing". In that way, instead of immediately returning the final value (which a synchronous method would do), Promises allow you to use an asynchronous method, get the final value, and queue up "next steps" that you want to run on the eventually-returned value, in the form of `.then()`s; you can tack callback functions onto Promises to handle what comes back from it. It's like saying

```
Promise me that you'll go to the supermarket
  .then(you'll make dinner with the ingredients you bought)
  .then(i'll do the washing up)
```

It is common practice to learn the `.then()` method with Promises. `.then` takes a callback function and returns *another* Promise (!) and you can do nifty things like console.log out the returned value, or pass it onto another function.

Promises actually come with a few more methods than this, though, which can be useful for building in a bit more robustness in error handling. Similarly to the rest of Javascript, you can use built-in, Promise-specific methods called `.catch()` and `.finally()`.

A Promise executes immediately and either resolves to a single value, or rejects with an error object. If the promise is rejected, the return value passes through any `.then`s and is picked up by the `.catch` (there is also a third state, 'pending', which is when it's still


Open in app Get started

This came in handy recently when handling the return from an API call to a database of census information. The `fetch` took in a criterion as entered by the user, then returned the gathered information to the frontend. I knew there would be trouble if the connection was interrupted, or the server wasn't connected.

```
API.getResultsByCriteria(UserInput)
  .then((people) => this.onResponseSuccess(people, selectedCategory))
  .catch(error => window.alert("Oops! Is your server disconnected?!"))
  .finally(() => this.setState({ loadingResults: false }))
```

[IMAGE DESCRIPTION: Javascript code which makes an API call, and handles the response with a .then(), .catch(), .finally() process]

I didn't do anything with the `error` object, and instead just manufactured a window alert, but made sure to implement a `.finally()` clause. This way, whatever happened, the part of state called `loadingResults` changed to false (necessary for an unmentioned part of the UI), and the code would not just stop working with no user feedback.

Interestingly, though `.finally()` takes a callback like the other two methods, it doesn't have access to (and therefore can't alter, in any way) the value that your Promise will resolve to.

Thanks to Robbie Heygate

About Help Terms Privacy

Get the Medium app

Open in app Get started