29th January 2020

Error handling with promises

Promise chains are great at error handling. When a promise rejects, the control jumps to the closest rejection handler. That's very convenient in practice.

For instance, in the code below the URL to `fetch` is wrong (no such site) and `.catch` handles the error:

```
fetch('https://no-such-server.blabla') // rejects
  .then(response => response.json())
  .catch(err => alert(err)) // TypeError: failed to fetch (the text may vary)
```

As you can see, the `.catch` doesn't have to be immediate. It may appear after one or maybe several `.then`.

Or, maybe, everything is all right with the site, but the response is not valid JSON. The easiest way to catch all errors is to append `.catch` to the end of chain:

```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  .then(response => response.json())
  .then(githubUser => new Promise((resolve, reject) => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  }))
  .catch(error => alert(error.message));
```

Normally, such `.catch` doesn't trigger at all. But if any of the promises above rejects (a network problem or invalid json or whatever), then it would catch it.

Implicit try…catch

The code of a promise executor and promise handlers has an "invisible `try..catch`" around it. If an exception happens, it gets caught and treated as a rejection.

For instance, this code:

```
new Promise((resolve, reject) => {
  throw new Error("Whoops!");
}).catch(alert); // Error: Whoops!
```

…Works exactly the same as this:

```
new Promise((resolve, reject) => {
  reject(new Error("Whoops!"));
}).catch(alert); // Error: Whoops!
```

The "invisible `try..catch`" around the executor automatically catches the error and turns it into rejected promise.

This happens not only in the executor function, but in its handlers as well. If we `throw` inside a `.then` handler, that means a rejected promise, so the control jumps to the nearest error handler.

Here's an example:

```
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  throw new Error("Whoops!"); // rejects the promise
}).catch(alert); // Error: Whoops!
```

This happens for all errors, not just those caused by the `throw` statement. For example, a programming error:

```
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  blabla(); // no such function
}).catch(alert); // ReferenceError: blabla is not defined
```

The final `.catch` not only catches explicit rejections, but also accidental errors in the handlers above.

Rethrowing

As we already noticed, `.catch` at the end of the chain is similar to `try..catch`. We may have as many `.then` handlers as we want, and then use a single `.catch` at the end to handle errors in all of them.

In a regular `try..catch` we can analyze the error and maybe rethrow it if it can't be handled. The same thing is possible for promises.

If we `throw` inside `.catch`, then the control goes to the next closest error handler. And if we handle the error and finish normally, then it continues to the next closest successful `.then` handler.

In the example below the `.catch` successfully handles the error:

```
// the execution: catch -> then
new Promise((resolve, reject) => {

  throw new Error("Whoops!");

}).catch(function(error) {

  alert("The error is handled, continue normally");

}).then(() => alert("Next successful handler runs"));
```

Here the `.catch` block finishes normally. So the next successful `.then` handler is called.

In the example below we see the other situation with `.catch`. The handler `(*)` catches the error and just can't handle it (e.g. it only knows how to handle `URIError`), so it throws it again:

```
// the execution: catch -> catch -> then
new Promise((resolve, reject) => {

  throw new Error("Whoops!");

}).catch(function(error) { // (*)

  if (error instanceof URIError) {
    // handle it
  } else {
    alert("Can't handle such error");

    throw error; // throwing this or another error jumps to the next catch
  }

}).then(function() {
  /* doesn't run here */
}).catch(error => { // (**)

  alert(`The unknown error has occurred: ${error}`);
  // don't return anything => execution goes the normal way

});
```

The execution jumps from the first `.catch` `(*)` to the next one `(**)` down the chain.

Unhandled rejections

What happens when an error is not handled? For instance, we forgot to append `.catch` to the end of the chain, like here:

```
new Promise(function() {
  noSuchFunction(); // Error here (no such function)
})
  .then(() => {
    // successful promise handlers, one or more
  }); // without .catch at the end!
```

In case of an error, the promise becomes rejected, and the execution should jump to the closest rejection handler. But there is none. So the error gets “stuck”. There’s no code to handle it.

In practice, just like with regular unhandled errors in code, it means that something has gone terribly wrong.

What happens when a regular error occurs and is not caught by `try..catch` ? The script dies with a message in the console. A similar thing happens with unhandled promise rejections.

The JavaScript engine tracks such rejections and generates a global error in that case. You can see it in the console if you run the example above.

In the browser we can catch such errors using the event `unhandledrejection` :

```
window.addEventListener('unhandledrejection', function(event) {
  // the event object has two special properties:
  alert(event.promise); // [object Promise] - the promise that generated the
  alert(event.reason); // Error: Whoops! - the unhandled error object
});

new Promise(function() {
  throw new Error("Whoops!");
}); // no catch to handle the error
```

The event is the part of the HTML standard.

If an error occurs, and there’s no `.catch` , the `unhandledrejection` handler triggers, and gets the `event` object with the information about the error, so we can do something.

Usually such errors are unrecoverable, so our best way out is to inform the user about the problem and probably report the incident to the server.

In non-browser environments like Node.js there are other ways to track unhandled errors.

Summary

- `.catch` handles errors in promises of all kinds: be it a `reject()` call, or an error thrown in a handler.
- We should place `.catch` exactly in places where we want to handle errors and know how to handle them. The handler should analyze errors (custom error classes help) and rethrow unknown ones (maybe they are programming mistakes).
- It’s ok not to use `.catch` at all, if there’s no way to recover from an error.
- In any case we should have the `unhandledrejection` event handler (for browsers, and analogs for other environments) to track unhandled errors and inform the user (and probably our server) about them, so that our app never “just dies”.

Tasks

Error in setTimeout

What do you think? Will the `.catch` trigger? Explain your answer.

```
new Promise(function(resolve, reject) {
  setTimeout(() => {
    throw new Error("Whoops!");
  }, 1000);
}).catch(alert);
```

solution

Previous lesson Next lesson

Share Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)