



Ruben Verborgh

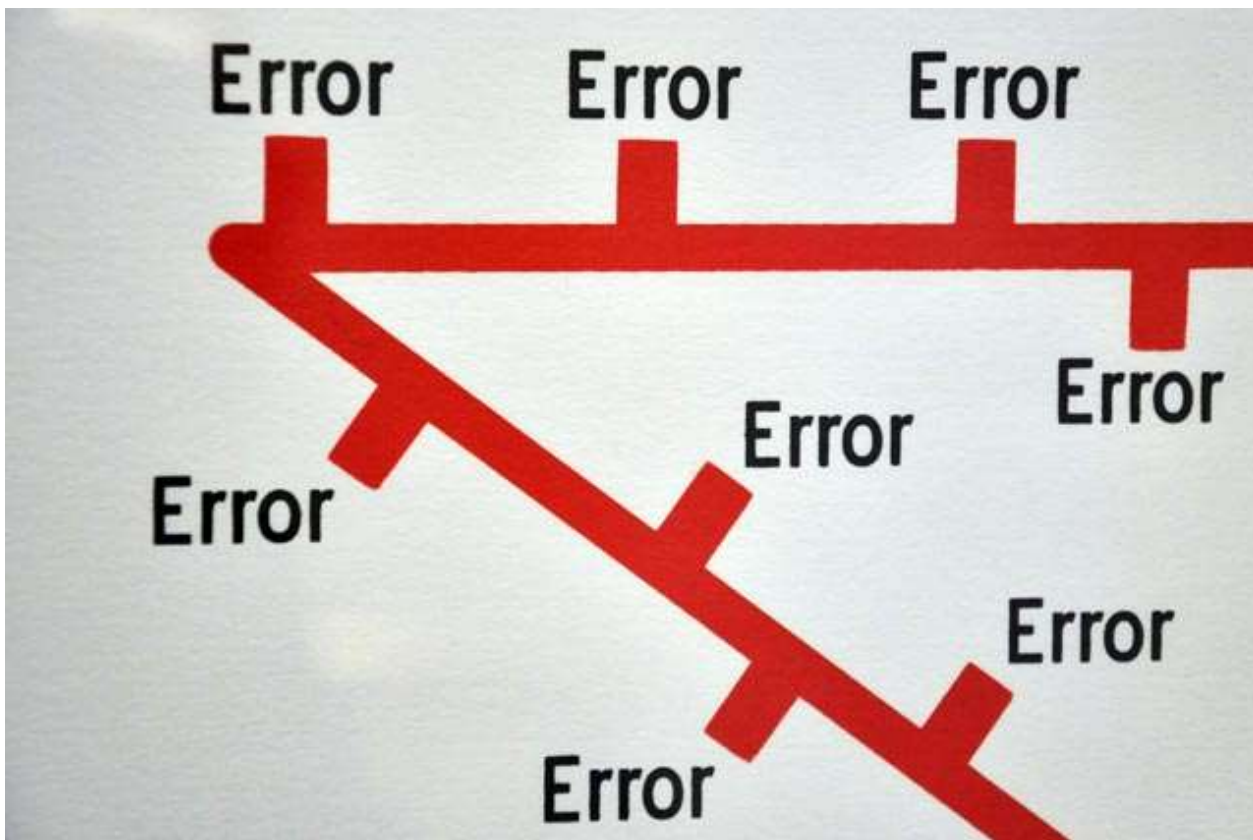
# Asynchronous error handling in JavaScript

Dealing with errors can be especially tricky in asynchronous situations.

**ANYTHING THAT CAN GO WRONG WILL GO WRONG, SO WE BETTER PREPARE OURSELVES. THE LESSONS WE'VE BEEN TAUGHT AS PROGRAMMERS to nicely throw and catch exceptions don't apply anymore in asynchronous environments. Yet asynchronous programming is on the rise, and things still can and therefore *will* go wrong. So what are your options to defend against errors and graciously inform the user when things didn't go as expected? This post compares different asynchronous error handling tactics for JavaScript.**

The easy case is when actions happen **synchronously**. Suppose you want to post a letter to a friend. Synchronous behavior is when you follow each step of the process and wait. So you pick up the envelope, put it in your pocket, ride with your bike to your friend's place, and deposit it in her letter box. If something goes wrong, such as you losing the envelope along the way, then you react as it happens by ringing the doorbell and apologizing.

An **asynchronous** way to do the same thing would be to call postal services. You hand over the letter to them, and they will do the steps for you, while you do something else. However, how will you know if things go wrong? After all, you don't witness the envelope sliding into the mailbox. Will the mailman call you on success or failure? Or will you call your friend to confirm successful arrival?



*The road to asynchronous success is paved with errors. How will you handle them? ©Nick J. Webb*

## Synchronous error handling

One of the earliest techniques that predates exceptions was to verify success depending on a function's **return value**.

```
function postLetter(letter, address) {  
  if (canSendTo(address)) {  
    letter.sendTo(address);  
    return true;  
  }  
  return false;  
}
```

You might have encountered this in C code. The caller thus inspects the value:

```
if (postLetter(myLetter, myAddress))  
  console.log("Letter sent.");  
else  
  console.error("Letter not sent.");
```

However, this is not convenient if the function also has to return an **actual value**. In that case, the caller would have to check whether the return value is legitimate or an error code. For that reason, **exceptions** have been invented.

```
function postLetter(letter, address) {  
  if (canSendTo(address)) {  
    letter.sendTo(address);  
    return letter.getTrackingCode();  
  }  
  throw "Cannot reach address " + address;  
}
```

The caller can then nicely **catch the exception** in a dedicated place.

```
try {
  var trackingCode = postLetter(myLetter, myAddress);
  console.log("Letter sent with code " + trackingCode);
}
catch (errorMessage) {
  console.error("Letter not sent: " + errorMessage);
}
```

Unfortunately, exceptions are only a **synchronous mechanism**, which is logical: in an asynchronous environment, the exception could be thrown when the handler block is already out of scope and thus meaningless.

## Error callbacks

One of the characteristics of asynchronous functions is that they **cannot immediately calculate their return value** (*otherwise, they wouldn't be asynchronous*). Instead, the return value is passed through a **callback function**. The same mechanism can pass an error value. One option is to use a **separate error callback**.

```
function postLetter(letter, address, onSuccess, onFailure) {
  if (canSendTo(address))
    letter.sendTo(address, function () {
      onSuccess(letter.getTrackingCode());
    });
  else
    onFailure("Cannot reach address " + address);
}
```

Note that `sendTo` is now also asynchronous here, calling the specified function after the letter has been send. The corresponding caller code looks like this:

```
postLetter(myLetter, myAddress,
  function (trackingCode) {
    console.log("Letter sent with code " + trackingCode);
  },
  function (errorMessage) {
    console.error("Letter not sent: " + errorMessage);
  });
```

It's a bit more verbose, but that's inherent to asynchronous programming with callbacks. An alternate solution is to stick to a single callback, adding an extra **error argument**. This is the approach many of the **Node.js APIs** take.

```
function postLetter(letter, address, callback) {
  if (canSendTo(address))
    letter.sendTo(address, function () {
      callback(null, letter.getTrackingCode());
    });
  else
    callback("Cannot reach address " + address);
}
```

If the first argument is **non-null**, an error has occurred. This can be seen in the caller:

```
postLetter(myLetter, myAddress,
  function (errorMessage, trackingCode) {
    if (errorMessage)
      return console.error("Letter not sent: " + errorMessage);
    console.log("Letter sent with code " + trackingCode);
  });
```

## Handling errors with promises

While callbacks provide a viable solution, they don't let us benefit from the added temporal flexibility that asynchronous programming offers. What if, at a certain point in time, we want to **change the performed action** in case of success or failure? Furthermore, it's a pity to **ignore the provided functionality of return values**. This is where **promises** come in. A **promise** is a **placeholder value** that can be returned by an asynchronous function. It acts as an **event emitter** that will indicate when the actual value is available, or when an error has occurred. Many implementations exist, such as **Q**, **when.js** and my own **promiscuous**. This example uses **jQuery** since many are familiar with this library (*although its promise implementation is **debated***).

The idea is that the asynchronous function creates a **deferred object** that provides the necessary functionality. Then, the function will **return the promise** that is created by the deferred object. The function **resolves** the deferred in case of success, or **rejects** it in case of failure.

```
function postLetter(letter, address) {
  var deferred = new $.Deferred();
  if (canSendTo(address))
    letter.sendTo(address, function () {
      deferred.resolve(letter.getTrackingCode());
    });
  else
    deferred.reject("Cannot reach address " + address);
}
```

```
    return deferred.promise();
  }
```

Note how the function’s signature is the same as the **synchronous one**. The return type, however, is different: it’s not a string but a **promise**. The caller can register success functions with `done` and failure functions with `fail`:

```
var trackingCodePromise = postLetter(myLetter, myAddress);
trackingCodePromise.done(function (trackingCode) {
  console.log("Letter sent with code " + trackingCode);
});
trackingCodePromise.fail(function (errorMessage) {
  console.log("Letter not sent: " + errorMessage);
});
```

Or, more compact and jQueryish:

```
$.when(postLetter(myLetter, myAddress))
  .then(function (trackingCode) {
    console.log("Letter sent with code " + trackingCode);
  },
  function (errorMessage) {
    console.log("Letter not sent: " + errorMessage);
  });
```

The fact that the latter code resembles the **error callback code** is **no coincidence**. Promises are in fact syntactic sugar. However, they can improve your code structure and readability, **increasing the maintainability** of your software.

## Domain-bound exceptions

The above two techniques are the classical ways to handle asynchronous errors. Node 0.8 has introduced the experimental **Domain API**, which allows the combination of asynchrony and exceptions. In fact, domains are a mechanism that allows to define **exception handlers as callbacks**.

Thereby, domains offer a **hybrid solution**: the function sends its return value through a **callback**, but errors are thrown through **exceptions**.

```
function postLetter(letter, address, callback) {
  if (!canSendTo(address))
    throw "Cannot reach address " + address;
  letter.sendTo(address, function () {
    callback(letter.getTrackingCode());
  });
}
```

The caller has to **create a domain** and **register an error handler**. Then, the calling code has to **run inside the domain**.

```
var postDomain = domain.create();
postDomain.on('error', function (errorMessage) {
  console.log("Letter not sent: " + errorMessage);
});
postDomain.run(function () {
  postLetter(myLetter, myAddress, function (trackingCode) {
    console.log("Letter sent with code " + trackingCode);
  });
});
```

Domains have other advantages and possibilities, such as **implicit and explicit binding**, but the details would take us too far here.

## Pick what you need

In the end, you are free to choose the mechanism that works best for you. I recommend promises for browser-based code, as UI-driven development tends to depend a lot on callbacks and deferreds allow a clean **separation between success and failure code**. In JavaScript server code, I never felt the need for deferreds. I tend to go for the **conventional Node.js callbacks**, although they can **automatically be converted into deferreds**. We still have to see whether the domain approach finds adoption, although I tend to use **asynchronous utility libraries** instead, as they ease nested callbacks and associated error handling.

**Try out the examples on GitHub** to discover the various error handling strategies. What’s **your favourite option**—and do you know **other techniques**?

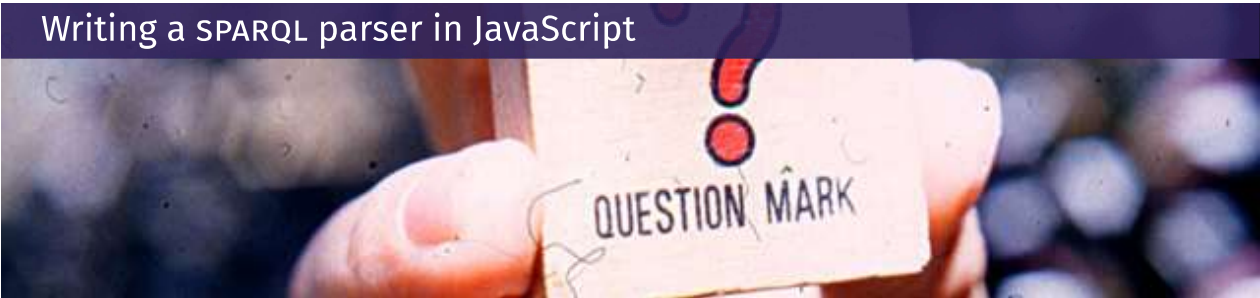
Ruben

31 December 2012

## 🔗 Discover more

Curious about *JavaScript*? There’s more!





Enjoyed this blog post? **Subscribe to the feed** for updates!

**Comment on this post**

13 Comments

Ruben Verborgh

1 Login

Recommend 4

Share

Sort by Oldest

Join the discussion...



OR SIGN UP WITH DISQUS ?

Name

- Paulo poiati • 5 years ago

Nice article.

| • Reply • Share ›
- Jim Alateras • 5 years ago

Great summary of the various mechanisms. I'm also in the same boat and use conventional nodejs callbacks for server side error handling.

4 | • Reply • Share ›
- Domenic Denicola • 5 years ago

Unfortunately, jQuery's promises fall down in exactly the error-handling use case. They're not a good example. See <https://gist.github.com/388...> for more details.

2 | • Reply • Share ›
- Ruben Verborgh 

Owner

 ➔ Domenic Denicola • 5 years ago

Hi [@Domenic Denicola](#), I see you have a strong opinion on promises :-)

Let's take a step back. Here, I'm just explaining the different methods of asynchronous error handling. Promises are one method, jQuery's implementation is one way to do it, and various people are familiar with this library.

1 | • Reply • Share ›
- Domenic Denicola ➔ Ruben Verborgh • 5 years ago

Yeah, and that's totally fair. But, it's a bit unfortunate to point to jQuery's implementation as an error handling solution, when in particular it is very bad at asynchronous error handling. Its promises are good for callback aggregation, but using them as an error handling solution will end in frustration.


1 | • Reply • Share ›
- Ruben Verborgh 

Owner

 ➔ Domenic Denicola • 5 years ago

I agree with your essay and hope that jQuery 2.0 will follow. However, I still think it offers the easiest introduction for beginners, but I have updated this post to reflect your concerns. Thanks a lot!

1 | • Reply • Share ›

 **Daniel Hölbling** • 5 years ago


Good post,

The main beauty of promises to me is that you can pass the promise to multiple sites that can then have their own error/success mechanic instead of having to extract the response at a central level..

I've too often had to write stuff like: `doFoo(function (err, response) { doSomething(response); doSomethingElse(response)' });` just to get one response value into another async thread to continue working..


With promises it's much easier to just take the promise and pass it off as even if it's already finished, hooking up to `.success()` will still yield the response that was generated when the promise was generated.

3 ^ | v • Reply • Share ›

 **Ruben Verborgh** Owner ➔ Daniel Hölbling • 5 years ago

I can't agree more! Promises make error handling a lot more flexible indeed.


^ | v • Reply • Share ›

 **Alex Tatumizer** • 5 years ago

Handling errors in asynchronous code is just a variant of much more general problem of point of control. Here's an attempt of more systemic solution:


<https://github.com/tatumize...>

^ | v • Reply • Share ›

 **Murvin** • 5 years ago

Good post. I have never used promise and good example. Yours is good and easy to understand. :)

^ | v • Reply • Share ›

 **Jason** • 5 years ago


I'm trying to create a working example of promises without using any libraries such as jQuery. I've created a hybrid of Crockford's code and your code above but it doesn't quite work, maybe someone can suggest how to make it work? Here's the code:

```
// Function that checks for valid address
function canSendTo(address) {
  if (address !== undefined) {
    return true;
  }
  return false;
}

// Define letter.sendTo, will need to take 2 parameters
// Also define letter.getTrackingCode for testing purposes
var letter = {
  sendTo : function(address, callback) {
    console.log('Sending letter to ' + address);
    callback(address);
  },
  getTrackingCode : function() {
    // ...
  }
};
```

see more


^ | v • Reply • Share ›

 **Ruben Verborgh** Owner ➔ Jason • 5 years ago

Hi Jason, I think the question here is: why reinvent the wheel? Good promise libraries exist, and they have been tested really well, so it's more likely to obtain solid code by using such a library. If size is a concern, [when.js](#) is under 1.3k and fast.

Can you try porting your code to such a library? If you want to discuss code, it's often more easy to create a [Gist](#), which has syntax highlighting and editing capabilities.

^ | v • Reply • Share ›

 **rotatopoti** • 10 months ago

Really nice article. A lot of articles show the callback function without showing how they are implemented.

^ | v • Reply • Share ›

