The Promises FAQ - addressing the most common questions and misconceptions about Promises

> This article was originally published at https://gist.github.com/joepie91/4c3a10629a4263a522e3bc4839a28c83. Nowadays Promises are more widely understood and supported, and it's not as relevant as it once was, but it's kept here for posterity.

> By the way, I'm available for tutoring and code review :)

You'll find a table of contents on your left.

1. What Promises library should I use?

That depends a bit on your usecase.

My usual recommendation is Bluebird - it's robust, has good error handling and debugging facilities, is fast, and has a well-designed API. The downside is that Bluebird will not correctly work in older browsers (think Internet Explorer 8 and older), and when used in Browserified/Webpacked code, it can sometimes add a lot to your bundle size.

ES6 Promises are gaining a lot of traction purely because of being "ES6", but in practice they are just *not very good*. They are generally lacking standardized debugging facilities, they are missing essential utilities such as Promise.try/promisify/promisifyAll, they cannot catch specific error types (this is a big robustness issue), and so on.

ES6 Promises can be useful in constrained scenarios (eg. older browsers with a polyfill, restricted non-V8 runtimes, etc.) but I would not generally recommend them.

There are many other Promise implementations (Q, WhenJS, etc.) - but frankly, I've not seen any that are an improvement over either Bluebird or ES6 Promises in their respective 'optimal scenarios'. I'd also recommend explicitly *against* Q because it is extremely slow and has a very poorly designed API.

In summary: Use Bluebird, unless you have a very specific reason not to. In those very specific cases, you probably want ES6 Promises.

2. How do I create a Promise myself?

Usually, you don't. Promises are not usually something you 'create' explicitly - rather, they're a *natural consequence* of chaining together multiple operations. Take this example:

```
function getLinesFromSomething() {
    return Promise.try(() => {
        return bhttp.get("http://example.com/something.txt");
    }).then((response) => {
        return response.body.toString().split("\n");
    });
}
```

In this example, all of the following *technically* result in a new Promise:

- `Promise.try(...)`
- `bhttp.get(...)`
- The synchronous value from the `.then` callback, which gets converted automatically to a resolved Promise (see question 5)

... but none of them are explicitly created as "a new Promise" - that's just the natural consequence of starting a chain with `Promise.try` and then returning Promises or values from the callbacks.

There is one example to this, where you *do* need to explicitly create a new Promise - when converting a different kind of asynchronous API to a Promises API, and even then you only need to do this if `promisify` and friends don't work. This is explained in question 7.

3. How do I use `new Promise`?

You don't, usually. In almost every case, you either need Promise.try, or some kind of promisification method. Question 7 explains how you should do promisification, and when you *do* need `new Promise`.

But when in doubt, don't use it. It's very error-prone.

4. How do I resolve a Promise?

You don't, usually. Promises are not something you need to 'resolve' manually - rather, you should just *return* some kind of Promise, and let the Promise library handle the rest.

There's one exception here: when you're manually promisifying a strange API using `new Promise`, you need to call `resolve()` or `reject()` for a successful and unsuccessful state, respectively. Make sure to read question 3, though - you should almost never actually *use* `new Promise`.

5. But what if I want to resolve a synchronous result or error?

You simply `return` it (if it's a result) or `throw` it (if it's an error), from your `.then` callback. When using Promises, synchronously returned values are automatically converted into a *resolved Promise*, whereas synchronously thrown errors are automatically converted into a *rejected Promise*. You don't need to use `Promise.resolve()` or `Promise.reject()`.

6. But what if it's at the start of a chain, and I'm not in a `.then` callback yet?

Using Promise.try will make this problem not exist.

7. How do I make this non-Promises library work with Promises?

That depends on what kind of API it is.

- **Node.js-style error-first callbacks:** Use Promise.promisify and/or Promise.promisifyAll to convert the library to a Promises API. For ES6 Promises, use the es6-promisify and es6-

promisify-all libraries respectively. In Node.js, `util.promisify` can also be used.

- **EventEmitters:** It depends. Promises are explicitly meant to represent an operation that succeeds or fails *precisely once*, so *most* EventEmitters cannot be converted to a Promise, as they will have *multiple* results. Some exceptions exist; for example, the `response` event when making a HTTP request - in these cases, use something like bluebird-events.
- **setTimeout:** Use `Promise.delay` instead, which comes with Bluebird.
- **setInterval:** Avoid `setInterval` entirely (this is why), and use a recursive `Promise.delay` instead.
- **Asynchronous callbacks with a single result argument, and no `err`:** Use promisify-simple-callback.
- **A different Promises library:** No manual conversion is necessary, as long as it is compliant with the Promises/A+ specification (and nearly every implementation is). Make sure to use Promise.try in your code, though.
- **Synchronous functions:** No manual conversion is necessary. Synchronous returns and throws are automatically converted by your Promises library. Make sure to use Promise.try in your code, though.
- **Something else not listed here:** You'll probably have to promisify it manually, using `new Promise`. Make sure to keep the code within `new Promise` as minimal as possible - you should have a function that *only* promisifies the API you intend to use, without doing *anything* else. All further processing should happen outside of `new Promise`, once you already have a Promise object.

8. How do I propagate errors, like with `if(err) return cb(err)`?

You don't. Promises will propagate errors automatically, and you don't need to do anything special for it - this is one of the benefits that Promises provide over error-first callbacks.

When using Promises, the *only* case where you need to `.catch` an error, is if you intend to handle it - and you should always *only* catch the types of error you're interested in.

These two Gists (step 1, step 2) show how error propagation works, and how to `.catch` specific types of errors.

9. How do I break out of a Promise chain early?

You don't. You use conditionals instead. Of course, specifically for *failure scenarios*, you'd still throw an error.

10. How do I convert a Promise to a synchronous value?

You can't. Once you write asynchronous code, all of the 'surrounding' code *also* needs to be asynchronous. However, you can just have a Promise chain in the 'parent code', and return the Promise from your own method.

For example:

```
function getUserFromDatabase(userId) {
    return Promise.try(() => {
        return database.table("users").where({id: userId}).get();
    }).then((results) => {
        if (results.length === 0) {
            throw new MyCustomError("No users found with that ID");
        } else {
            return results[0];
        }
    });
}

/* Now, to *use* that getUserFromDatabase function, we need to have another Promise chain: */

Promise.try(() => {
    // Here, we return the result of calling our own function. That return value is a Promise.
    return getUserFromDatabase(42);
}).then((user) => {
    console.log("The username of user 42 is:", user.username);
});
```

(If you're not sure what Promise.try is or does, this article will explain it.)

11. How do I save a value from a Promise outside of the callback?

You don't. See question 10 above - you need to use Promises "all the way down".

12. How do I access previous results from the Promise chain?

In some cases, you might need to access an *earlier* result from a chain of Promises, one that you don't have access to anymore. A simple example of this scenario:

```
'use strict';

// ...

Promise.try(() => {
    return database.query("users", {id: req.body.userId});
}).then((user) => {
    return database.query("groups", {id: req.body.groupId});
}).then((group) => {
    res.json({
        user: user, // This is not possible, because `user` is not in scope anymore.
        group: group
    });
});
```

This is a fairly simple case - the `user` query and the `group` query are completely independent, and they can be run at the same time. Because of that, we can use `Promise.all` to run them in parallel, and return a combined Promise for *both* of their results:

```
'use strict';

// ...

Promise.try(() => {
    return Promise.all([
        database.query("users", {id: req.body.userId}),
```

```
        database.query("groups", {id: req.body.groupId})
    ]);
}).spread((user, group) => {
    res.json({
        user: user, // Now it's possible!
        group: group
    });
});
```

Note that instead of `.then`, we use `.spread` here. Promises only support a *single* result argument for a `.then`, which is why a Promise created by `Promise.all` would resolve to an array of `[user, group]` in this case. However, `.spread` is a Bluebird-specific variation of `.then`, that will automatically "unpack" that array into multiple callback arguments. Alternatively, you can use ES6 object destructuring to accomplish the same.

Now, the above example assumes that the two asynchronous operations are *independent* - that is, they can run in parallel without caring about the result of the other operation. In some cases, you will want to use the results of two operations that are *dependent* - while you still want to use the results of both at the same time, the second operation also needs the result of the first operation to work.

An example:

```
'use strict';

// ...

Promise.try(() => {
    return getDatabaseConnection();
}).then((databaseConnection) => {
    return databaseConnection.query("users", {id: req.body.id});
}).then((user) => {
    res.json(user);

    // This is not possible, because we don't have `databaseConnection` in scope anymore:
    databaseConnection.close();
});
```

In these cases, rather than using `Promise.all`, you'd *add a level of nesting* to keep something in scope:

```
'use strict';

// ...

Promise.try(() => {
    return getDatabaseConnection();
}).then((databaseConnection) => {
    // We nest here, so that `databaseConnection` remains in scope.

    return Promise.try(() => {
        return databaseConnection.query("users", {id: req.body.id});
    }).then((user) => {
        res.json(user);

        databaseConnection.close(); // Now it works!
    });
});
```

Of course, as with any kind of nesting, you should do it sparingly - and only when necessary for a situation like this. Splitting up your code into small functions, with each of them having a *single* responsibility, will prevent trouble with this.

Revision #1
Created 2024-12-11 12:12:14 UTC by joepie91
Updated 2024-12-11 18:44:19 UTC by joepie91