Javascript Promises

then

reject / resolve

Samy Pessé

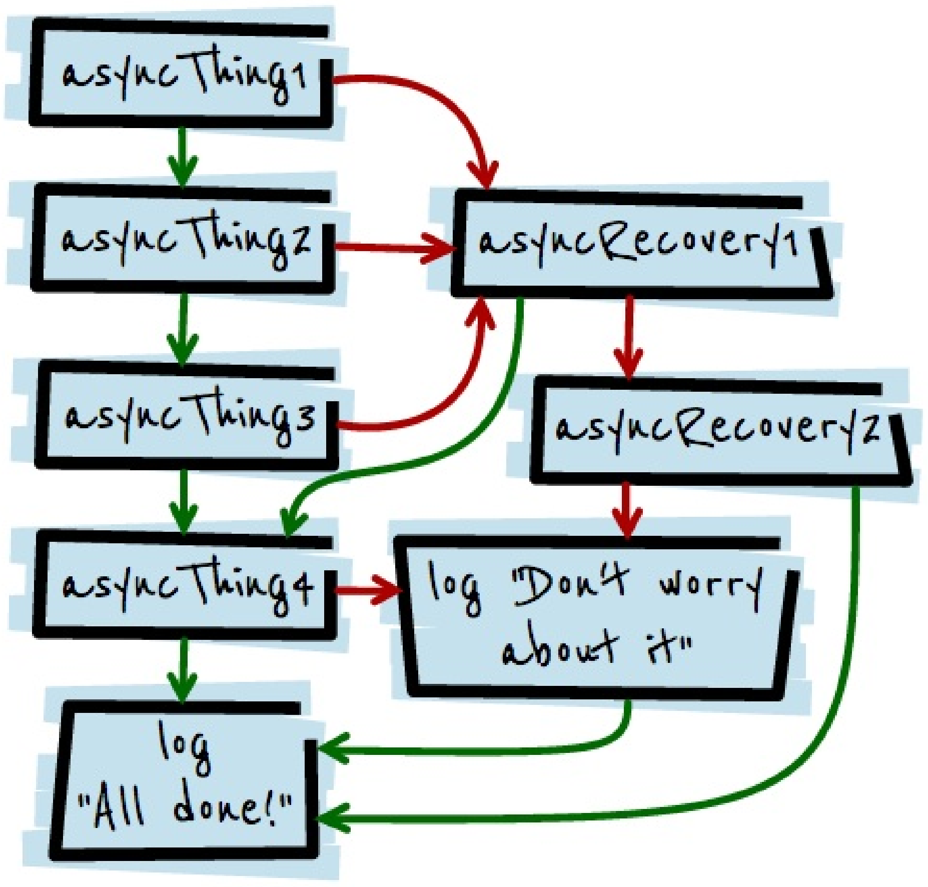
Published with GitBook

Table of Contents

1. Introduction
2. What are Promises?
3. Chaining
4. Error handling
5. Parallelism and sequencing
6. Librairies
7. API Reference

Javascript Promises

In this book, you'll learn how to use promises in Javascript, and why you should use it.



We'll talk about native promises but also about using library such as Q; in Node.js and client-side.

This book will contain quizzes so that you can test your knowledges.

What are Promises?

Lets start by discovering what are promises and why we should use it.

Javascript and Async

JavaScript is single threaded, meaning that two bits of script cannot run at the same time, they have to run one after another. In browsers, JavaScript shares a thread with a load of other stuff. What that stuff is differs from browser to browser, but typically JavaScript is in the same queue as painting, updating styles, and handling user actions (such as highlighting text and interacting with form controls). Activity in one of these things delays the others.

Luckily, Javascript (Node.js and Broweer) has a lot of asynchronous API. The way it exposes asynchronous programming to the application logic is via events or callbacks.

In event-based asynchronous APIs, developers register an event handler for a given object (e.g. HTML Element or other DOM objects) and then call the action. The browser or node.js will perform the action usually in a different thread, and trigger the event in the main thread when appropriate.

For example in the browser for doing an HTTP request (**event based**):

```
// Create the XHR object to do GET to /data resource
var xhr = new XMLHttpRequest();
xhr.open("GET","data",true);

// register the event handler
xhr.addEventListener('load',function(){
  if(xhr.status === 200){
     alert("We got data: " + xhr.response);
  }
},false)

// perform the work
xhr.send();
```

And in Node.js for reading a file (**callback based**):

```
var fs = require("fs");

fs.readFile('/etc/passwd', function (err, data) {
    // An error occured
    if (err) throw err;

    // Result:
    console.log(data);
});
```

Events and Callbacks aren't always the best way

Events are great for things that can happen multiple times on the same object (keyup, touchstart etc). With those events you don't really care about what happened before you attached the listener.

But when it comes to async success/failure, you need to use callback based APIs. And if your application logic starts to do more things, it become ugly really fast.

For example, if you want to read the content of a file, **then** send it to a server and **then** write the headers result in a file:

```
var fs = require("fs");
var http = require("http");

var myOperation = function(input, output, callback) {
    fs.readFile(input, function (err, data) {
        // An error occured
        if (err) callback(err);

        http.post("http://www.google.com/index.html", {
            body: data
        }, function(res) {
            if (res.statusCode != 200) {
                callback(new Error("Invalid http request"));
            }

            var content;

            try {
                content = JSON.stringify(res.header);
            } catch(e) {
                callback(e);
            }

            fs.writeFile(output, content, function (err) {
                // An error occured
                if (err) callback(err);

                callback(null, "done!");
            });

        }).on('error', function(e) {
          callback(e);
        });
    });
};

myOperation("./input.txt", "./output.txt", function(err) {
    if (err) throw err;

    console.log("done!");
});
```

With Promises

Promises allows you to write asynchronous code in a more synchronous fashion.

Basically, A Promise object represents a value that may not be available yet, but will be resolved at some point in future. For example, if you use the Promise API to make an asynchronous call to a remote web service you will create a Promise object which represents the data that will be returned by the web service in future. The caveat being that the actual data is not available yet. It will become available when the request completes and a response comes back from the web service. In the meantime the Promise object acts like a proxy to the actual data. Further, you can attach callbacks to the Promise object which will be called once the actual data is available.

A promise is in one of three different states:

- **pending** - The initial state of a promise.
- **fulfilled** - The state of a promise representing a successful operation.
- **rejected** - The state of a promise representing a failed operation.

Once a promise is fulfilled or rejected, it is immutable (i.e. it can never change again).

So, at their most basic, promises are a bit like event listeners except:

- A promise can only succeed or fail once. It cannot succeed or fail twice, neither can it switch from success to failure or vice versa
- If a promise has succeeded or failed and you later add a success/failure callback, the correct callback will be called, even though the event took place earlier

This is extremely useful for async success/failure, because you're less interested in the exact time something became available, and more interested in reacting to the outcome.

If Node.js APIs were nativelly promises-based, the operation from before will look like:

```
var fs = require("fs");
var http = require("http");

var myOperation = function(input, output) {
   return fs.readFile(input)
   .then(function (data) {
      return http.post("http://www.google.com/index.html", {
         body: data
      })
   })
   .then(function(res) {
      if (res.statusCode != 200) {
         throw new Error("Invalid http request");
      }

      return JSON.stringify(res.header);
   })
   .then(function(data) {
      return fs.writeFile(output, data);
   });
};

myOperation("./input.txt", "./output.txt")
.then(function() {
   console.log("done!");
}, function(err) {
   console.log("Error:", err);
});
```

The keyword in this code is **then**.

Constructing a promise

You'll need to construct promises by hand, since not all APIs supports Promises.

We use `new Promise` to construct the promise. We give the constructor a factory function which does the actual work. This function is called immediately with two arguments. The first argument fulfills the promise and the second argument rejects the promise. Once the operation has completed, we call the appropriate function.

Here is a promise based of the node.js `fs.readFile`:

```
function readFile(filename){
   return new Promise(function (fulfill, reject) {
      fs.readFile(filename, function (err, res) {
         if (err) reject(err);
         else fulfill(res);
      });
   });
}
```

So we can use it like this:

```
readFile("./test.txt")
.then(function(data) {
   // Succees !
}, function(err) {
   // Error :(
});
```

Chaining

Chaining is what is so great about promises.

Since Promises capture the notion of asynchronicity in an object, we can chain them, map them, have them run in parallel or sequential, all kinds of useful things. Code like the following is very common with Promises:

```
getSomeData()
.then(filterTheData)
.then(processTheData)
.then(displayTheData);
```

`getSomeData` is returning a Promise, as evidenced by the call to `then()`, but the result of that first then must also be a Promise, as we call `then()` again (and yet again!) That's exactly what happens, if we can convince `then()` to return a Promise, things get more interesting.

> then() always returns a Promise

`then()` takes 2 arguments, one for success, one for failure (or fulfill and reject, in promises-speak), each will return a promises.

Transforming values

Chaining promises is useful for transforming values, each `.then(func)` return a promise resolved with the returned value of `func`. For example:

```
var promise = new Promise(function(resolve, reject) {
   resolve(1);
});

promise
.then(function(val) {
   console.log(val); // 1
   return val + 2;
})
.then(function(val) {
   console.log(val); // 3
   return val * 3;
});
.then(function(val) {
   console.log(val); // 9
});
```

As a practical example, chaining of promises is great for processing data from an HTTP request: get the content - parse the content - filter the content.

Example with the GitHub API

Here is an example using the GitHub API and promises, that will display the list of repositories of the most followed GitHub user.

For this example, we'll consider the method `get(url)` as doing the HTTP GET method to url and returning a promise that'll be resolved with the plain text content.

```
var BASE_URL = "https://api.github.com";

function getReposMostFollowedUser() {
    // Serach users and ordr by followers
    return get(BASE_URL + "/search/users?q=followers:>500&sort=followers&order=desc")

    // Parse the text content as JSON
    .then(JSON.parse)

    // Extract first user
    .then(function(users) {
        if (users.length == 0) throw "No user to return";

        return users[0];
    })

    // Get repositories list for the user
    .then(function(user) {
        return get(BASE_URL + "/users/"+user.login+"/repos")
    })

    // Parse the text content as JSON
    .then(JSON.parse);
};
```

Then we can easily call the method `getReposMostFollowedUser()` that will return a promise.

```
getReposMostFollowedUser()
.then(function(repos) {
    console.log("Repos:", repos);
});
```

Error handling

As we saw earlier, `then` takes two arguments, one for success, one for failure (or fulfill and reject, in promises-speak):

```
get('story.json')
.then(function(response) {
   console.log("Success!", response);
}, function(error) {
   console.log("Failed!", error);
});
```

.fail, .catch

Some librairies implement the method `.fail` (or `.catch`) which is equivalent to `.then(undefined, func)`, for example:

```
get('story.json')
.then(function(response) {
   return JSON.parse(response);
})
.then(function(response) {
   console.log("Success!", response);
})
.fail(function(error) {
   console.log("Failed!", error);
});
```

Rejection order

The following codes are not equivalent:

```
get('story.json')
.then(function(response) {
  console.log("Success!", response);
}, function(error) {
  console.log("Failed!", error);
});
```

and

```
get('story.json')
.then(function(response) {
  console.log("Success!", response);
})
.fail(function(error) {
  console.log("Failed!", error);
});
```

The difference is subtle, but extremely useful. Promise rejections skip forward to the next `then` with a rejection callback (or `fail`, since it's equivalent). With `then(func1, func2)`, `func1` or `func2` will be called, never both. But with `then(func1).fail(func2)`, both will be called if `func1` rejects, as they're separate steps in the chain.

Javascript Exceptions

Rejections happen when a promise is explicitly rejected, but also implicitly if an error is thrown in the constructor callback. Let's take a look at the previous example:

```
// Step1: get the json content
get('story.json')

// Step2: parse the json content
.then(function(response) {
    return JSON.parse(response);
})

// Step3: show the parsed results
.then(function(response) {
    console.log("Success!", response);
})

// Handle errors
.fail(function(error) {
    console.log("Failed!", error);
});
```

The Javascript method JSON.parse can throw an exception (`SyntaxError`) if the string is not valid JSON. This exception (if occurs) will be handled by the `.fail` .

Basically, the `.then(func)` is encapsulating the call to `func` with a `try { func(resolved) } catch (e) { // return rejected promise }` .

Parallelism and sequencing

Librairies

API Reference