Javascript Promises

then
reject / resolve

Samy Pessé

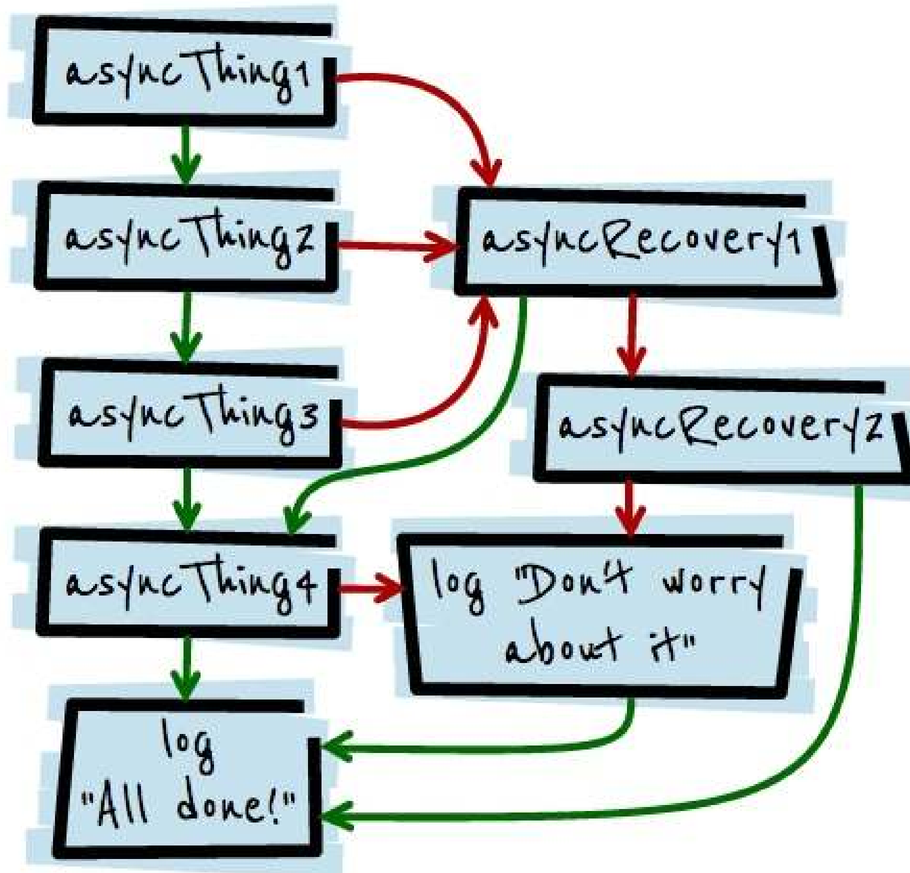
Published
with GitBook

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

Click here to download full PDF material