

Not quite what you are looking for? You may want to try:

- Simple radial gradient implementation with XAML sample application for X11 (but without the need for any external library)
- IIFE – Immediately Invoked Function Expressions

[highlights off](#)

12,503,146 members (63,900 online)

[Sign in](#)



[articles](#) [Q&A](#) [forums](#) [lounge](#)

 *

Promises in JavaScript - Past, Present and the Future



Nitij, 26 Jun 2016

CPOL

Rate this:



4.50 (2 votes)

An article about the history, current trends and the future of JavaScript Promises

Introduction

Today we cannot imagine writing Javascript code which is not asynchronous in nature for any decent web application. Whether its the service calls, animations or even simple stuff like displaying time on a web page we must utilize JavaScript's asynchronous elements. So what is exactly synchronous and asynchronous programming? The very simple difference is that synchronous code must wait for its previous action to complete before moving on to execute next action.

Synchronous code usually runs in a single thread. Async code must execute in separate threads. When the async JavaScript code runs the browser takes care of managing the threads for us in simple scenarios. We always need to write async code when we have to wait for result before moving on to execute any task; for example when we make a Get request on the server then we have to wait until the results are returned back to us and we can show them on the page. This requirement to wait leads to the need of callbacks.

A callback is simply a piece of code which gets called after the result of any async request is retrieved. Now callbacks can be of two types: success and failure. When the result is successfully received then a success callback should be executed. When there is any problem in executing the asynchronous code then a failure callback is executed. Both success and failure callbacks could be same or different functions in JavaScript which depends on our preference.

When we are dealing with async code and callbacks then promises come into the picture. **A promise is something which we receive in the future and can have several states; the most common are Promise Fulfilled and Promise Rejected.**

As per github/kriskowal

Quote:

If a function cannot return a value or throw an exception without blocking, it can return a promise instead. A promise is an object that represents the return value or the thrown exception that the function may eventually provide. A promise can also be used as a proxy for a remote object to overcome latency.

You can read the Commonjs specification for promises over here:

<http://wiki.commonjs.org/wiki/Promises/A>

The Past

XML Http Request

Before we had jQuery's promises the usual way to assign callbacks to async service calls was by using **XHR** callbacks. An Xhr callback can be assigned to success and failed responses from the server using designated status codes.

Following is the basic implementation of xhr object. In the simplest of scenarios we will just create a new xhr object and then we will have to send the request. Before sending the request we can assign a callback based on the ready state and the status of our request.

[Hide](#) [Copy Code](#)

```
if (window.XMLHttpRequest)
{ // code for IE7+, Firefox, Chrome, Opera, Safari
    xmlhttp = new XMLHttpRequest();
}
else
{ // code for IE6, IE5
    xmlhttp = new ActiveXObject('Microsoft.XMLHTTP');
}
xmlhttp.onreadystatechange = function ()
{
    if (xmlhttp.readyState == 4 && xmlhttp.status == 200)
    {
        successCallback();
    }
}
xmlhttp.open('GET', url, true);
xmlhttp.send();
```

The ready state can have flowing values:

- 0 = Unsent
- 1 = Opened
- 2 = Headers Received
- 3 = Loading
- 4 = Completed

XHR status can have following values:

- 200 = Okay
- 404 = Page Not Found

In the current version we can assign event listeners to xhr events.

[Hide](#) [Copy Code](#)

```
var xhr = new XMLHttpRequest();
xhr.addEventListener("progress", progress);
xhr.addEventListener("load", complete);
xhr.addEventListener("error", failed);
xhr.addEventListener("abort", cancelled);
```

The callbacks could be normal functions:

[Hide](#) [Copy Code](#)

```
function completed(data) {
    document.getElementById('container').innerHTML = data.Text;
}
```

Callbacks in jQuery

jQuery's `$.ajax()` had success and error callbacks which have been widely used to manage latency problems with the server in returning the data. Its implementation is simple; two JavaScript properties to which we assign functions which later get called in case of successfully getting the result or if there is any error.

[Hide](#) [Copy Code](#)

```
$.ajax({
    method: "POST",
    url: "some.php",
    data: { name: "John", location: "Boston" },
    error: function (xhr, error) {
        //handle the error
    },
    success: function (data) {
        //handle the returned data
    }
});
```

```
}); }
```

The Present

.then(fn)

Fast forward to today and we see .then() everywhere in the JS code. I can bet that you will see this implementation in every mid to large sized application utilizing a decent JS framework. As the name implies .then() is used to execute a callback code after a response is received from an async code execution.

Following is the jQuery implementation of .then() function using a Deferred Object:

[Hide](#) [Copy Code](#)

```
$ .when ($.ajax("test.aspx")).then(function (data, textStatus, jqXHR) {  
    alert(jqXHR.status);  
});
```

Chaining Callbacks

There can be numerous cases when based on the result returned from one async code block execution we need to make another async request. In such cases we must chain the callbacks one after the another. To implement the callback chaining we need a special kind of object which can store the callback information and can then queue their execution.

[Hide](#) [Copy Code](#)

```
$ .when ($.ajax("someUrl.aspx"))  
    .then(function (data, textStatus, jqXHR) {  
        $.ajax("someOtherUrl.aspx");  
    })  
    .then(function (data, textStatus, jqXHR) {  
        $.ajax("someOtherUrl.aspx");  
    })  
    .then(function (data, textStatus, jqXHR) {  
        $.ajax("yetAnotherUrl.aspx");  
    })  
);
```

Deferred Object

This object can facilitate callback chaining and their queued execution. A deferred object exposes several functions which could be used to chain success and failure callbacks. Other features may include getting the current state of deferred object and rejecting and resolving with a specific context.

Promise implementation in jQuery

jQuery returns deferred objects from several APIs which involve some kind of async code execution. For instance \$.ajax() and many animation function which returns deferred objects and we can chain further callbacks to those objects.

[Hide](#) [Copy Code](#)

```
$("#container").css("color", "red").slideUp(2000).slideDown(2000);
```

Other JS Code Libraries

There are other code libraries out there available for us to use if don't want to use the inherent promise code structure of any framework. These libraries work more or less in the same way, that is to chain callbacks one after another for success and failed responses. The most prominent one is \$q.

<https://github.com/kriskowal/q>

AngularJS has implemented \$q to handle async code execution and provides the \$q dependency as an in-built service which we can use in our custom service factories.

The motive of \$q or any other such deferred object library is to fix the horizontally growing callback chains and convert them into linear code statements.

[Hide](#) [Copy Code](#)

```
function callback1 (data) {
    function callback2(data) {
        function callback3(data) {
            function callback4(data) {
                //and so on
            }
        }
    }
}
```

The above code could be converted into a linear chain using deferred objects to chain promises.

[Hide](#) [Copy Code](#)

```
someFunctionReturningPromise()
  .then(callBack1)
  .then(callBack2)
  .then(callBack3)
  .then(callBack4);
```

The Future

When we talk about the future of Promises in JavaScript then it would not be wrong to say that the future is already here. The only thing holding the latest promise implementations to come into practice is the widespread use by the developer community. That too is changing as more and more companies and teams are switching over to the ES6/ES7 coding standards.

ES6 Promises

ES6 has a standard specification for a Promise object which could be used to handle the future results of an operation. The syntax of Promise constructor is as follows:

[Hide](#) [Copy Code](#)

```
new Promise( /* executor */ function(resolve, reject) { ... } );
```

In the above we must provide an executor function. An executor function gets executed when we call the then() method of the Promise object. The resolve and reject arguments are functions which we pass into the executor to handle the future outcome of the task.

Following is a very simple example which shows how it could be used:

[Hide](#) [Copy Code](#)

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <div id="container"></div>
  <script>
    var myPromise = new Promise(function (resolve, reject) {
      setTimeout(function () {
        resolve('Hello');
      }, 5000);
    });
    myPromise.then(function (message) {
      document.getElementById('container').innerHTML = message;
    });

  </script>
</body>
</html>
```

The above code will print a message of our choice in the container <div> element after 5 seconds. This syntax is still not compatible with most browsers so you must use a polyfill for older versions if you decide to use it in your application.

async Functions

You might have already written methods with the `async` keyword in c# language. Methods/Functions with the `async` keyword uses `await` keyword in the body to wait for the result of an asynchronous code execution. If the result is received then the next set of code executes and if it does not then an error is caught by the catch block. This type of implementation eliminates the need to write separate callback functions to chain in an `async` request/response task execution.

[Hide](#) [Copy Code](#)

```
<script>
  async function GET(url) {
    var result = await getResults(url);
    return result;
  }
</script>
```

Async functions could not be used in browsers yet without transpiling the JS code using a suitable transpiler. A suitable replacement for `async-await` in JavaScript could be generator functions.

[Hide](#) [Copy Code](#)

```
<script>
  function* someGenerator(url){
    yield get(url);
  }

  console.log(someGenerator('account/1').next().value);
</script>
```

Unfortunately the support for generators is still very limited and it would be a while before we could see its widespread use in browsers.

The advanced implementations of promises in JavaScript has still a long way to go before things could become stable. But the good news is that things are advancing very fast or us. There is already support for such features in Node.js environment. For the browsers there are several good transpilers out there like Babel which we could use to write and take advantage of the new syntax.

What I have seen over past couple of years is that so many Devs are still not very clear about what promises are and how they could be leveraged not just in `async` server calls but in so many different areas.

Custom Implementation

If you are feeling ambitious then you can even try to implement your own promise object. I created a custom deferred object implemented some time back which supports chaining multiple callbacks:

<https://github.com/Nitij/NoviceJsPromise>

I would advise to use a good promise library instead of writing our own implementations because its risky to write custom code for such complicated specifications and its always better to use code modules which are highly tested and have a widespread use and support by the developer community.

That is it for now folks. There is still a lot which could be covered on promises; if you are a newbie then treat this article as a way to start off into working with `async` code in JavaScript. If you are an expert then this might act as a refresher for you. Feel free to give meaningful comments or ask any question.

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

Share

[EMAIL](#)

[TWITTER](#)

About the Author



Nitij

 Software Developer
India 

Just a regular guy writing regular code 😊

I have over 5 years of experience of working in several client and server technology stacks like Asp.Net Mvc, Web Api, JavaScript, jQuery, Angular, Require, React, Handlebars and so on.

I have recently started a YouTube channel having videos on different programming topics. I would advise to check it out as you might find something useful over there.

[My YouTube Channel](#)

If you are interested in working with me then drop a message on my website homepage and maybe together we can create something awesome 😊

[My Homepage](#)

Cheers!

Comments and Discussions

You must [Sign In](#) to use this message board.

[First](#) [Prev](#) [Next](#)

[Read more about ES6 Promises](#)

Gerd Wagner 27-Jun-16 2:44

For a tutorial on ES6 Promises, see our blog article [Learn JavaScript/ES6 Promises by Examples](#).

[Sign In](#) · [View Thread](#) · [Permalink](#)

[Refresh](#)

1

 General  News  Suggestion  Question  Bug  Answer  Joke  Praise  Rant  Admin

[Permalink](#) | [Advertise](#) | [Privacy](#) | [Terms of Use](#) | [Mobile](#)
Web02 | 2.8.160919.1 | Last Updated 26 Jun 2016

Select Language ▾
Layout: [fixed](#) | [fluid](#)

Article Copyright 2016 by Nitij

Everything else Copyright © [CodeProject](#), 1999-2016