JavaScript goes Asynchronous (and it's awesome).

David Catuhe, 4 Nov 2015 CPOL

Microsoft Program Manager David Cathuhe shares an overview of asynchronous code within ECMAScript: what it is, how it works, and ways it can improve your workflow when working with JavaScript.

JavaScript has come a long way since its early versions and thanks to all efforts done by TC39 (The organization in charge of standardizing JavaScript (or **ECMAScript** to be exact)) we now have a modern language that is used widely.

One area within **ECMAScript** that received vast improvements is **asynchronous code**. You can learn more about asynchronous programming here if you're a new developer. Fortunately we've included these changes in Windows 10's new Edge browser—check out the Microsoft Edge change log.

Among all these new features, let's specifically focus on "ES2016 **Async Functions**" behind the **Experimental Javascript** features flag and take a journey through the updates and see how ECMAScript can improve your currently workflow.

First stop: ECMAScript 5 – Callbacks city

ECMAScript 5 (and previous versions as well) are all about callbacks. To better picture this, let's have a simple example that you certainly use more than once a day: executing a XHR request.

Hide Shrink Copy Code

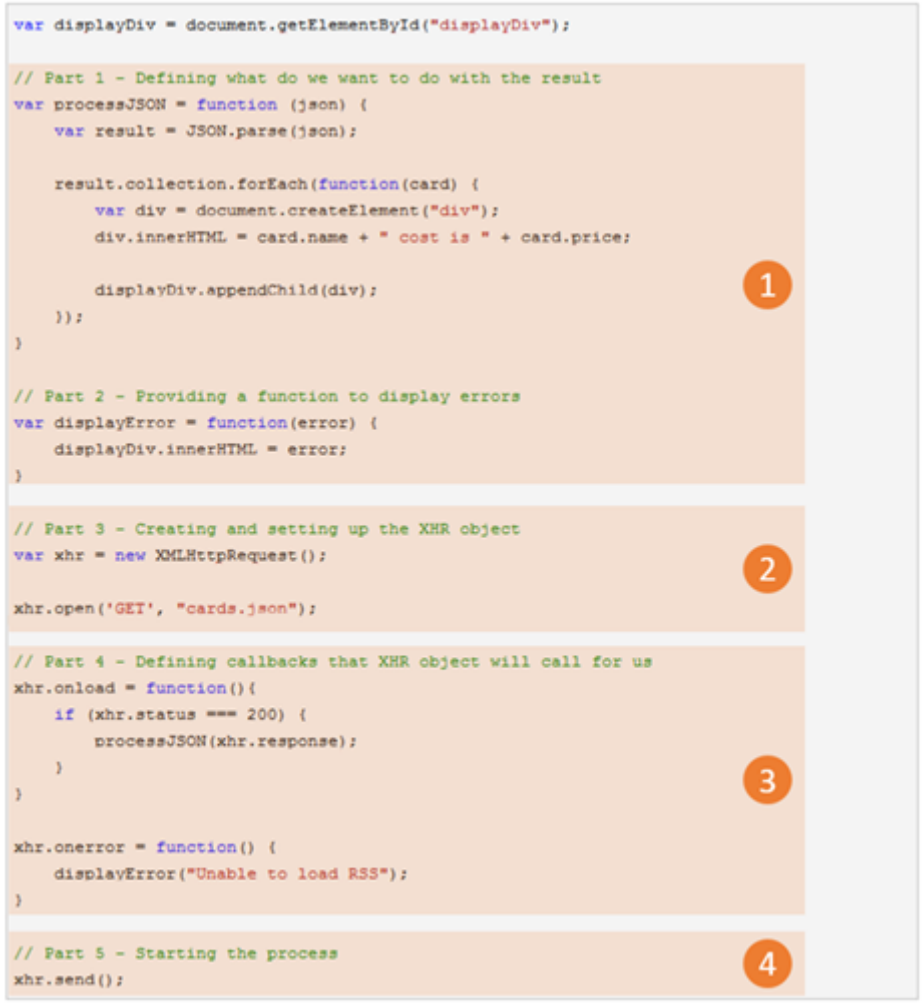
```
var displayDiv = document.getElementById("displayDiv");

// Part 1 - Defining what do we want to do with the result
var processJSON = function (json) {
    var result = JSON.parse(json);

    result.collection.forEach(function(card) {
        var div = document.createElement("div");
        div.innerHTML = card.name + " cost is " + card.price;

        displayDiv.appendChild(div);
    });
}

// Part 2 - Providing a function to display errors
var displayError = function(error) {
```

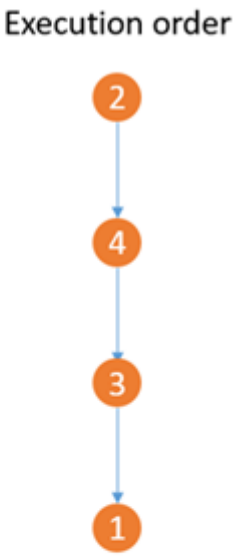
```
    displayDiv.innerHTML = error;
}

// Part 3 - Creating and setting up the XHR object
var xhr = new XMLHttpRequest();

xhr.open('GET', "cards.json");

// Part 4 - Defining callbacks that XHR object will call for us
xhr.onload = function(){
    if (xhr.status === 200) {
        processJSON(xhr.response);
    }
}

xhr.onerror = function() {
    displayError("Unable to load RSS");
}

// Part 5 - Starting the process
xhr.send();
```

Established JavaScript developers will note how familiar this looks since XHR callbacks are used all the time! It's simple and fairly straight forward: the developer creates an XHR request and then provides the callback for the specified XHR object.

In contrast, callback complexity comes from the execution order which is not linear due to the inner nature of asynchronous code:

```
var displayDiv = document.getElementById("displayDiv");

// Part 1 - Defining what do we want to do with the result
var processJSON = function (json) {
    var result = JSON.parse(json);

    result.collection.forEach(function(card) {
        var div = document.createElement("div");
        div.innerHTML = card.name + " cost is " + card.price;

        displayDiv.appendChild(div);
    });
}

// Part 2 - Providing a function to display errors
var displayError = function(error) {
    displayDiv.innerHTML = error;
}

// Part 3 - Creating and setting up the XHR object
var xhr = new XMLHttpRequest();

xhr.open('GET', "cards.json");

// Part 4 - Defining callbacks that XHR object will call for us
xhr.onload = function(){
    if (xhr.status === 200) {
        processJSON(xhr.response);
    }
}

xhr.onerror = function() {
    displayError("Unable to load RSS");
}

// Part 5 - Starting the process
xhr.send();
```





The "callbacks hell" can even be worse when using another asynchronous call inside of your own callback.

Second stop: ECMAScript 6 – Promises city

ECMAScript 6 is gaining momentum and Edge is has leading support with 88% coverage so far.

Among a lot of great improvements, **ECMAScript 6** standardizes the usage of *promises* (formerly known as futures).

According to MDN, a *promise* is an object which is used for deferred and asynchronous computations. A *promise* represents an operation that hasn't completed yet, but is expected in the future. Promises are a way of organizing asynchronous operations in such a way that they appear synchronous. Exactly what we need for our XHR example.

Promises have been around for a while but the good news is that now you don't need any library anymore as they are provided by the browser.

Let's update our example a bit to support *promises* and see how it could improve the readability and maintainability of our code:

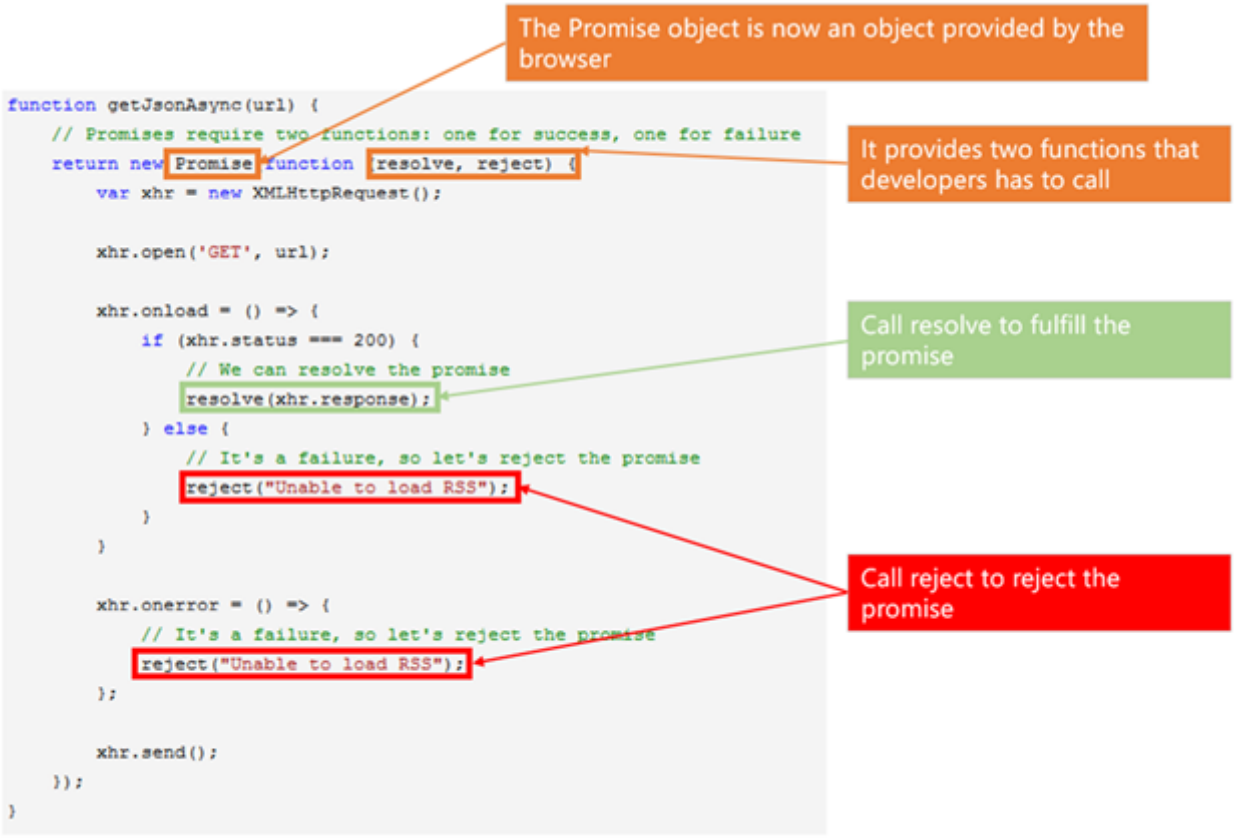
Hide Shrink Copy Code

```
var displayDiv = document.getElementById("displayDiv");

// Part 1 - Create a function that returns a promise
function getJsonAsync(url) {
    // Promises require two functions: one for success, one for failure
    return new Promise(function (resolve, reject) {
        var xhr = new XMLHttpRequest();

        xhr.open('GET', url);

        xhr.onload = () => {
            if (xhr.status === 200) {
                // We can resolve the promise
                resolve(xhr.response);
            } else {
                // It's a failure, so let's reject the promise
                reject("Unable to load RSS");
            }
        }

        xhr.onerror = () => {
            // It's a failure, so let's reject the promise
            reject("Unable to load RSS");
        };

        xhr.send();
    });
}

// Part 2 - The function returns a promise
// so we can chain with a .then and a .catch
getJsonAsync("cards.json").then(json => {
    var result = JSON.parse(json);

    result.collection.forEach(card => {
        var div = document.createElement("div");
        div.innerHTML = `${card.name} cost is ${card.price}`;

        displayDiv.appendChild(div);
    });
}).catch(error => {
    displayDiv.innerHTML = error;
});
```

You may have noticed a lot of improvements here. Let's have a closer look.

Creating the promise

In order to "promisify" (sorry but I'm French so I'm allowed to invent new words) the old XHR object, you need to create a `Promise` object:



Using the promise

Once created, the *promise* can be used to chain asynchronous calls in a more elegant way:

So now we have (from the user standpoint):

- Get the promise **(1)**

- Chain with the success code **(2 and 3)**
- Chain with the error code **(4)** like in a try/catch block

What's interesting is that chaining *promises* are easily called using *.then().then(),* etc.

Side node: *Since JavaScript is a modern language, you may notice that I've also used syntax sugar from* ***ECMAScript 6*** *like template strings or arrow functions.*

Terminus: ECMAScript 7 – Asynchronous city

Finally, we've reached our destination! We are almost in the future, but thanks to Edge's rapid development cycle, the team is able to introduce a bit of **ECMAScript 7**with **async functions** in the latest build!

Async functions are a syntax sugar to improve the language-level model for writing asynchronous code.

Async functions are built on top of ECMAScript 6 features like generators. Indeed, generators can be used jointly with promises to produce the same results but with much more user code

We do not need to change the function which generates the promise as async functions work directly with promise.

We only need to change the calling function:

Hide Copy Code

```
// Let's create an async anonymous function
(async function() {
    try {
        // Just have to await the promise!
        var json = await getJsonAsync("cards.json");
        var result = JSON.parse(json);

        result.collection.forEach(card => {
            var div = document.createElement("div");
            div.innerHTML = `${card.name} cost is ${card.price}`;

            displayDiv.appendChild(div);
        });
    } catch (e) {
        displayDiv.innerHTML = e;
    }
})();
```

This is where magic happens. This code looks like a regular synchronous code with a perfectly linear execution path:


```
// Let's create an async anonymous function
(async function() {
    try {
        // Just have to await the promise!
        var json = await getJsonAsync("cards.json");
        var result = JSON.parse(json);

        result.collection.forEach(card => {
            var div = document.createElement("div");
            div.innerHTML = `${card.name} cost is ${card.price}`;

            displayDiv.appendChild(div);
        });
    } catch (e) {
        displayDiv.innerHTML = e;
    }
})();
```

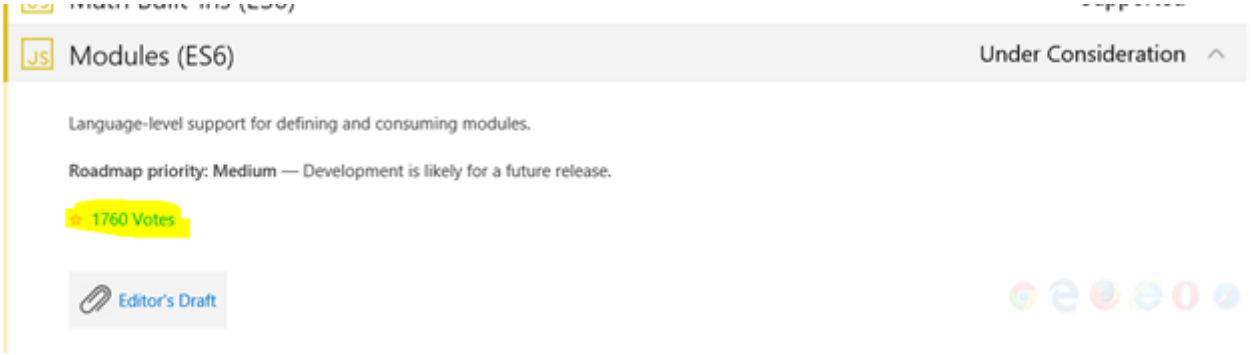
Quite impressive, right?

And the good news is that you can even use async functions with arrow functions or class methods.

Going further

If you want more detail on how we implemented it in Chakra, please check the official post on the Microsoft Edge blog. You can also track the progress of various browsers implementation of **ECMAScript 6** and **7** using Kangax's website.

Feel free also to check our JavaScript roadmap as well! Please, do not hesitate to give us your feedback and support your favorite features by using the vote button:



Thanks for reading and we're eager to hear your feedback and ideas!

More hands-on with Web Development

This article is part of the web development series from Microsoft tech evangelists and engineers on practical JavaScript learning, open source projects, and interoperability best practices including Microsoft Edge browser and the new EdgeHTML rendering engine.

We encourage you to test across browsers and devices including Microsoft Edge – the default browser for Windows 10 – with free tools on dev.modern.IE:

- Scan your site for out-of-date libraries, layout issues, and accessibility

- Use virtual machines for Mac, Linux, and Windows
- Remotely test for Microsoft Edge on your own device
- Coding Lab on GitHub: Cross-browser testing and best practices

In-depth tech learning on Microsoft Edge and the Web Platform:

- Microsoft Edge Web Summit 2015 (what to expect with the new browser, new supported web platform standards, and guest speakers from the JavaScript community)
- Woah, I can test Edge & IE on a Mac & Linux! (from Rey Bango)
- Advancing JavaScript without Breaking the Web (from Christian Heilmann)
- The Edge Rendering Engine that makes the Web just work (from Jacob Rossi)
- Unleash 3D rendering with WebGL (from David Catuhe including the vorlon.JS and babylonJS projects)
- Hosted web apps and web platform innovations (from Kevin Hill and Kiril Seksenov including the manifold.JS project)

More free cross-platform tools & resources for the Web Platform:

- Visual Studio Code for Linux, MacOS, and Windows
- Code with node.JS and free trial on Azure

License

Share

About the Author

David Catuhe

United States

David Catuhe is a Principal Program Manager at Microsoft focusing on web development. He is author of the babylon.js framework for building 3D games with HTML5 and WebGL. Read his blog on MSDN or follow him @deltakosh on Twitter.

Comments and Discussions

Coming from a C# background, this is a bliss

Reply · Email · View Thread · Permalink · Bookmark

Great writeup new
Chris Maunder 8-Nov-15 20:21

Thanks David - now all we need to do is wait for all the browsers to fully implement ES6.

What would be interesting is any article laying out best practices on how to use these features now while downgrading gracefully for older browsers.

---

cheers
Chris Maunder

Reply · Email · View Thread · Permalink · Bookmark

My vote of 5 new
Santhk 5-Nov-15 4:51

Good article

Reply · Email · View Thread · Permalink · Bookmark

Refresh 1

General News Suggestion Question Bug Answer Joke Praise Rant Admin