

Hiring? Toptal handpicks [top JavaScript engineers](#) to suit your needs.

- [Start hiring](#)
- [Log in](#)
- [Top 3%](#)
- [Why](#)
- [Clients](#)
- [Enterprise](#)
- [Partners](#)
- [Community](#)
- [Blog](#)
- [About Us](#)
- [Start hiring](#)
- [Apply as a Developer](#)
- [Login](#)
- - Questions?
 - [Contact Us](#)
 -
 -
 -

[Hire a developer](#)

Asynchronous JavaScript: From Callback Hell to Async and Await

[View all articles](#)



by [Demir Selmanovic](#) - JavaScript Developer @ [Toptal](#)

[#AsyncAwait](#) [#Asynchronous](#) [#JavaScript](#)

- 200shares





One of the keys to writing a successful web application is being able to make dozens of AJAX calls per page.

This is a typical asynchronous programming challenge, and how you choose to deal with asynchronous calls will, in large part, make or break your app, and by extension potentially your entire startup.

Synchronizing asynchronous tasks in JavaScript was a serious issue for a very long time.

This challenge is affecting back-end [developers using Node.js](#) as much as front-end developers using any JavaScript framework. Asynchronous programming is a part of our everyday work, but the challenge is often taken lightly and not considered at the right time.

A Brief History of Asynchronous JavaScript

The first and the most straightforward solution came in the form of *nested functions as callbacks*. This solution led to something called *callback hell*, and too many applications still feel the burn of it.

Then, we got *Promises*. This pattern made the code a lot easier to read, but it was a far cry from the Don't Repeat Yourself (DRY) principle. There were still too many cases where you had to repeat the same pieces of code to properly manage the application's flow. The latest addition, in the form of `async/await` statements, finally made asynchronous code in JavaScript as easy to read and write as any other piece of code.

Let's take a look at the examples of each of these solutions and reflect on the evolution of asynchronous programming in JavaScript.

To do this, we will examine a simple task that performs the following steps:

1. Verify the username and password of a user.
2. Get application roles for the user.
3. Log application access time for the user.

Approach 1: Callback Hell (“The Pyramid of Doom”)

The ancient solution to synchronize these calls was via nested callbacks. This was a decent approach for simple asynchronous JavaScript tasks, but wouldn't scale because of an issue called [callback hell](#).



The code for the three simple tasks would look something like this:

```
const verifyUser = function(username, password, callback){
  dataBase.verifyUser(username, password, (error, userInfo) => {
    if (error) {
      callback(error)
    }else{
      dataBase.getRoles(username, (error, roles) => {
        if (error){
          callback(error)
        }else {
          dataBase.logAccess(username, (error) => {
            if (error){
              callback(error);
            }else{
              callback(null, userInfo, roles);
            }
          })
        }
      })
    }
  })
}
```

```
    }  
  })  
};
```

Each function gets an argument which is another function that is called with a parameter that is the response of the previous action.

Too many people will experience brain freeze just by reading the sentence above. Having an application with hundreds of similar code blocks will cause even more trouble to the person maintaining the code, even if they wrote it themselves.

This example gets even more complicated once you realize that a `database.getRoles` is another function that has nested callbacks.

```
const getRoles = function (username, callback){  
  database.connect((connection) => {  
    connection.query('get roles sql', (result) => {  
      callback(null, result);  
    })  
  });  
};
```

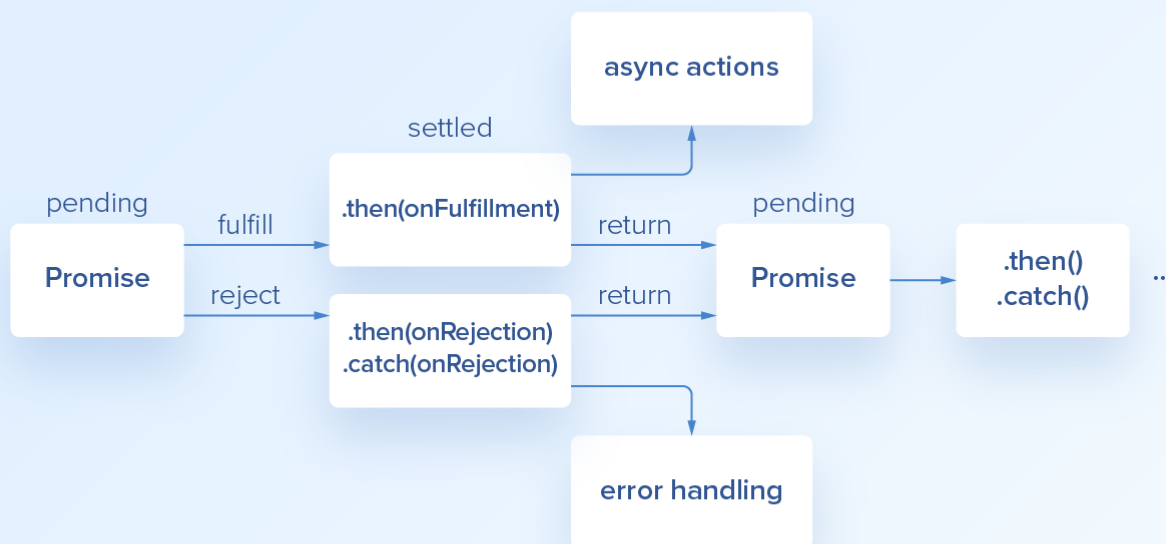
In addition to having code that is difficult to maintain, the DRY principle has absolutely no value in this case. Error handling, for example, is repeated in each function and the main callback is called from each nested function.

More complex asynchronous JavaScript operations, such as looping through asynchronous calls, is an even bigger challenge. In fact, there is no trivial way of doing this with callbacks. This is why JavaScript Promise libraries like [Bluebird](#) and [Q](#) got so much traction. They provide a way to perform common operations on asynchronous requests that the language itself doesn't already provide.

That's where native JavaScript Promises come in.

JavaScript Promises

[Promises](#) were the next logical step in escaping callback hell. This method did not remove the use of callbacks, but it made the chaining of functions straightforward and [simplified the code](#), making it much easier to read.



With Promises in place, the code in our asynchronous JavaScript example would look something like this:

```
const verifyUser = function(username, password) {
  database.verifyUser(username, password)
    .then(userInfo => dataBase.getRoles(userInfo))
    .then(rolesInfo => dataBase.logAccess(rolesInfo))
    .then(finalResult => {
      //do whatever the 'callback' would do
    })
    .catch((err) => {
      //do whatever the error handler needs
    });
};
```

To achieve this kind of simplicity, all of the functions used in the example would have to be *Promisified*. Let's take a look at how the `getRoles` method would be updated to return a Promise:

```
const getRoles = function (username){
  return new Promise((resolve, reject) => {
```

```

    database.connect((connection) => {
        connection.query('get roles sql', (result) => {
            resolve(result);
        })
    });
});
};

```

We have modified the method to return a Promise, with two callbacks, and the Promise itself performs actions from the method. Now, resolve and reject callbacks will be mapped to Promise.then and Promise.catch methods respectively.

You may notice that the getRoles method is still internally prone to the pyramid of doom phenomenon. This is due to the way database methods are created as they do not return Promise. If our database access methods also returned Promise the getRoles method would look like the following:

```

const getRoles = new function (userInfo) {
    return new Promise((resolve, reject) => {
        database.connect()
            .then((connection) => connection.query('get roles sql'))
            .then((result) => resolve(result))
            .catch(reject)
    });
};

```

Approach 3: Async/Await

JavaScript is asynchronous by default. This might be reason why it took so long to get synchronous-looking code that runs properly in JavaScript. But, better late than never! The pyramid of doom was significantly mitigated with the introduction of Promises. However, we still had to rely on callbacks that are passed on to .then and .catch methods of a Promise.

Promises paved the way to one of the coolest improvements in JavaScript. [ECMAScript 2017](#) brought in syntactic sugar on top of Promises in JavaScript in the form of async and await statements.

They allow us to write Promise-based code as if it were synchronous, but without blocking the main thread, as this code sample demonstrates:

```

const verifyUser = async function(username, password){
    try {
        const userInfo = await dataBase.verifyUser(username, password);
        const rolesInfo = await dataBase.getRoles(userInfo);
        const logStatus = await dataBase.logAccess(userInfo);
        return userInfo;
    }catch (e){
        //handle errors as needed
    }
};

```

Awaiting Promise to resolve is allowed only within async functions which means that verifyUser had to be defined using async function.

However, once this small change is made you can await any Promise without additional changes in other methods.

Async - A Long Awaited Resolution of a Promise

Async functions are the next logical step in the evolution of asynchronous programming in JavaScript. They will make your code much cleaner and easier to maintain. Declaring a function as `async` will ensure that it always returns a `Promise` so you don't have to worry about that anymore.

Why should you start using the JavaScript `async` function today?

1. The resulting code is much cleaner.
2. Error handling is much simpler and it relies on `try/catch` just like in any other synchronous code.
3. Debugging is much simpler. Setting a breakpoint inside a `.then` block will not move to the next `.then` because it only steps through synchronous code. But, you can step through `await` calls as if they were synchronous calls.

Understanding the Basics

What are `async` and `await`?

`Async/await` statements are syntactic sugar created on top of JavaScript Promises. They allow us to write Promise-based code as if it were synchronous, but without blocking the main thread.

What is callback hell?

In JavaScript, callback hell is an anti-pattern in code that happens as a result of poor structuring of asynchronous code. It is usually seen when programmers try to force a visual top-down structure in their asynchronous callback-based JavaScript code.

What are JavaScript promises?

A promise in JavaScript is like a placeholder value that is expected to eventually resolve into the final successful result value or reason for failure.

About the author



[View full profile »](#)

[Hire the Author](#)

[Demir Selmanovic, Bosnia and Herzegovina](#)

member since May 23, 2014

[Microsoft SQL Server](#) [Web App Development](#) [AJAX](#) [Express.js](#) [JavaScript](#) [C#](#) [jQuery](#) [Windows](#) [Google Glass](#) [+3 more](#)

Demir is a developer and project manager with over 15 years of professional experience in a wide range of software development roles. He excels as a solo developer, team member, team leader, or manager of multiple distributed teams. He works closely with clients to define ideas and deliver products. [\[click to continue...\]](#)

[Hiring? Meet the Top 10 Freelance JavaScript Developers for Hire in April 2018](#)

21 Comments

Toptal

 Login ▾

 Recommend 13

 Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name



Peter Edache • 3 months ago

Nice insight.

7 ^ | v • Reply • Share ›



iweczek • 3 months ago

Under Approach 3, I think you meant to say "JavaScript is synchronous by default." and "why it took so long to get asynchronous-looking code"...

5 ^ | v • Reply • Share ›



Eki Eqbal • 3 months ago

Next time someone asks me about the differences between Async/Await and JS Promises I am pointing them here. You did a very nice job in your write up, keep it up and thanks again for sharing. As a side note, [@Demir Selmanovic](#) has already written a book about ES9, It's currently sealed up. In two years, the JS community is going to open the book to see if the language design team got it right. Cheers

5 ^ | v • Reply • Share ›



Demir Selmanovic Mod ➔ Eki Eqbal • 3 months ago

Thanks Eki !!! :)

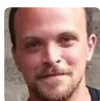
^ | v • Reply • Share ›



Nick McCrea Mod • 3 months ago

Great article Demir! Just what I was looking for!

^ | v • Reply • Share ›



Esteban • 3 months ago

Excelent job

^ | v • Reply • Share ›



Leonard Lepadatu • 3 months ago

The idea of this kind of article is not new. Gil Tayar made this three years ago in hebrew, then in english. Please fell free to watch:

Callback Hell Awaiting ES6 Heaven - Promises (Hebrew)



see more

^ | v • Reply • Share ›



Eduardo Campver • 3 months ago

Hi there, just something I consider worth adding: the use of generator functions and promises

together. It lays right between Promises and async/await. There is this package called `co-router` which allows you to use generator functions as route handlers in Express, then wherever a Promise was returned you could just use `yield asyncCall(args)`, sort of like `await`, but not quite though. Oh and thanks for the great post ;)

^ | v • Reply • Share ›



Demir Selmanovic Mod ➔ Eduardo Campver • 3 months ago

You are welcome, glad you liked the post :)

I had my doubts about adding generators. I decided to keep them out because I personally don't think they will gain that much traction comparing to what Promises gave us. And now that async/await is available (pretty much) I expect that many people will not even get into Generators. Still, I might be wrong so thanks for pointing it here so the readers will at least hear about them.

^ | v • Reply • Share ›



Eduardo Campver ➔ Demir Selmanovic • 3 months ago

I definitely wouldn't recommend using generators over async/await, it was just a bit of syntactic sugar back then before the "better syntactic sugar" came into play. As soon as I was able to, I replaced all use of `function*` and `yield` for `async function` and `await` respectively.

^ | v • Reply • Share ›



Joachim • 3 months ago

Just a quick note, your second example with promise-returning database methods can be simplified to:

```
const getRoles = new function (userInfo) {
  return database.connect()
    .then((connection) => connection.query('get roles sql'));
};
```

In general, code inside a new Promise() that simply passes resolve and reject to an inner Promise is always a superfluous wrapper.

^ | v • Reply • Share ›



Demir Selmanovic Mod ➔ Joachim • 3 months ago

Hey, thanks for the tip. Yeah, I'm aware that the code could have been made more concise. But the purpose of the post and the samples is to illustrate differences and progress from callback hell onward to the people that are not yet sure about "what the hell are Promises". So I chose to use the "more obvious" solution.

1 ^ | v • Reply • Share ›



Joachim ➔ Demir Selmanovic • 3 months ago

Fair enough. In that case, I think the call to forward to resolve should either be

```
.then(resolve)
```

or

```
.then(result => resolve(result))
```

depending on what you think is clearer.

^ | v • Reply • Share ›



Demir Selmanovic Mod ➔ Joachim • 3 months ago

Agreed. Better now? :)

^ | v • Reply • Share ›



Joachim ➔ Demir Selmanovic • 3 months ago

Yes, that's what I meant...

^ | v • Reply • Share ›



The Master ➔ Joachim • 3 months ago

I don't think Demir Selmanovic gives a ..block about your quick note !

^ | v • Reply • Share ›



Demir Selmanovic Mod ➔ The Master • 3 months ago

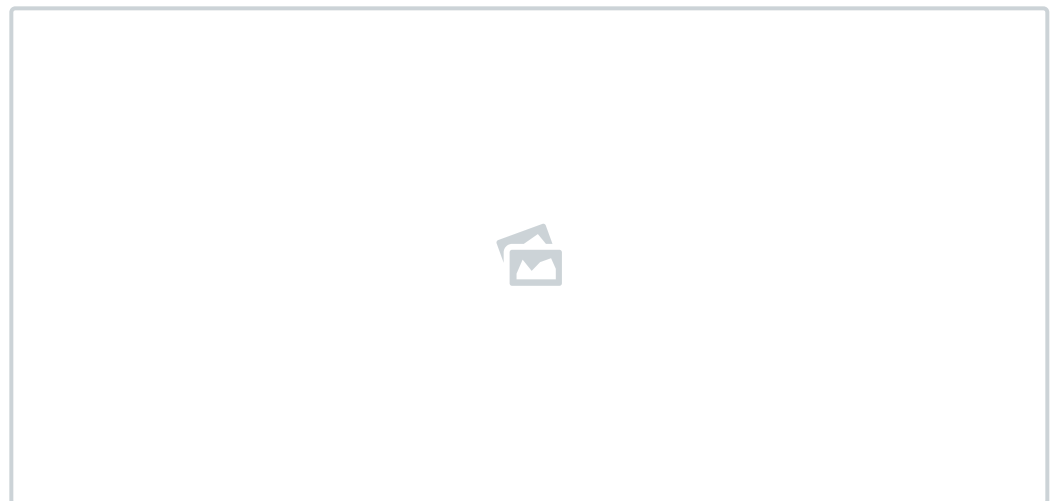
Well, I actually gave a paragraph :)

1 ^ | v • Reply • Share ›



The Master ➔ Demir Selmanovic • 3 months ago

Yes **@Demir Selmanovic** , you did !



1 ^ | v • Reply • Share ›



Randy Casburn • 3 months ago

Nice brief. May I recommend that you consider changing your WrapUp header? This "Wrapping Up: Async/Await vs. Promises" sounds as if the two ideas are competing while in fact they are complimentary. Would not want a novice to be confused. You have nailed to idea home though - Async/Await is a great enhancement to Promises.

^ | v • Reply • Share ›



Hi Randy, thanks for the tip. We have updated the header to something more interesting. I believe it is better now.



Check your inbox to confirm subscription. You'll start receiving posts after you confirm.

-


-



-



-



Microservice Communication: A Spring Integration Tutorial with

[Redis](#)8 days ago[4 Go Language Criticisms](#)10 days ago[One-click Login with Blockchain:](#)[A MetaMask Tutorial](#)12 days ago[Machine Learning Video Analysis: Identifying Fish](#)15 days ago[The Comprehensive Guide to JavaScript Design Patterns](#)16 days ago[Python Logging: An](#)[In-Depth Tutorial](#)22 days ago

Relevant Technologies

- [JavaScript](#)
- [Node.js](#)
- [AngularJS](#)
- [Backbone.js](#)
- [Front-End](#)

About the author

[Demir Selmanovic](#)

JavaScript Developer

Demir is a developer and project manager with over 15 years of professional experience in a wide range of software development roles. He excels as a solo developer, team member, team leader, or manager of multiple

distributed teams. He works closely with clients to define ideas and deliver products.

[Hire the Author](#)

Toptal connects the [top 3%](#) of freelance talent all over the world.

Toptal Developers

- [Android Developers](#)
- [AngularJS Developers](#)
- [Back-End Developers](#)
- [C++ Developers](#)
- [Data Scientists](#)
- [DevOps Engineers](#)
- [Ember.js Developers](#)
- [Freelance Developers](#)
- [Front-End Developers](#)
- [Full Stack Developers](#)
- [HTML5 Developers](#)
- [iOS Developers](#)
- [Java Developers](#)
- [JavaScript Developers](#)
- [Machine Learning Engineers](#)
- [Magento Developers](#)
- [Mobile App Developers](#)
- [.NET Developers](#)
- [Node.js Developers](#)
- [PHP Developers](#)
- [Python Developers](#)
- [React.js Developers](#)
- [Ruby Developers](#)
- [Ruby on Rails Developers](#)
- [Salesforce Developers](#)
- [Scala Developers](#)
- [Software Developers](#)
- [Unity or Unity3D Developers](#)
- [Web Developers](#)
- [WordPress Developers](#)

[See more freelance developers](#)

[Learn how enterprises benefit from Toptal experts.](#)

Join the Toptal community.

[Hire a developer](#)

or

[Apply as a Developer](#)

Highest In-Demand Talent

- [iOS Developer](#)
- [Front-End Developer](#)
- [UX Designer](#)
- [UI Designer](#)
- [Financial Modeling Consultants](#)
- [Interim CFOs](#)

About

- [Top 3%](#)
- [Clients](#)
- [Freelance Developers](#)
- [Freelance Designers](#)
- [Freelance Finance Experts](#)
- [About Us](#)

Contact

- [Contact Us](#)
- [Press Center](#)
- [Careers](#)
- [FAQ](#)

Social

- [Facebook](#)
- [Twitter](#)
- [Google+](#)
- [LinkedIn](#)

[Toptal](#)

Hire the top 3% of freelance talent

- © Copyright 2010 - 2018 Toptal, LLC
- [Privacy Policy](#)
- [Website Terms](#)

[Home](#) › [Blog](#) › [Asynchronous JavaScript: From Callback Hell to Async and Await](#)