**Callbacks in JavaScript -** [Popa Vlad](https://medium.com/@popa.vlad94?source=post_page-----551cd3a58f3a--------------------------------) Feb 3, 2019

Callbacks, or higher order functions, are available in many programming languages, for instance, as function-valued arguments (e.g. Python), function pointers (e.g. C++), and lambda expressions (e.g. Lisp). In this paper we study JavaScript callbacks since JavaScript is the dominant language for building web applications. Foe example, a recent survey of more than 26.000 developers conducted by Stack Overflow found that JavaScript is the most-used programming language. The callback language feature in JavaScript is an important factor to its success. For instance, JavaScript callbacks are used to responsively handle events on the client-side by executing functions asynchronously. And, in Node.js, a popular JavaScript-based framework, callbacks are used on the server-side to service multiple concurrent client requests.

Although callbacks are a key JavaScript feature, they have not received much attention in the research community. I think that this is a critical omission as the usage of callbacks is an important factor in developer comprehension and maintenance of JavaScript code.

var db = require('somedatabaseprovider');  
http.get('/recentposts', (req, res) => {  
 db.openConnection('host', creds, (err, con) => {  
 res.param['posts'].forEach((post) => {  
 conn.query('select \* from users where id=' + post['user'], (err, results) => {  
 conn.close();  
 res.send(results[0]);  
 });  
 });  
 });  
});// Listing 1.

The snipet code from above illustrates three common challenges with callbacks. First, the three callbacks in this example (on lines 2, 3 and 5) are *anonymous*. This makes the callbacks difficult to reuse and to understand as they are not descriptive. Second, the callbacks in the example are *nested*to three levels, making it challenging to reason about the flow of control in code. Finally, in line 5 there is a call to **conn.query**, which invokes the second callback parameter *asynchronously*. That is, the execution of the inner-most anonymous function (lines 6–7) is deferred until some later time. As a result, most of the complexity in this small example rests on extensive use of callbacks; in particular, the use of anonymous, nested, and asynchronous callbacks.

Though the issues outlined above have not been studied in detail, they are well-known to developers. Searching for “callback hell” on the web brings up many articles with best practices on callback usage in JavaScript. For example, a prominent problem with asynchronous callbacks in JavaScript is error-handling. The built-in try/catch mechanism does not work with asynchronous callbacks. In response, the JavaScript developer community has, over time, arrived at a best practice known as “error-first protocol”. The intent is to reserve the first argument in a callback for communicating errors. However, as this is a best practice and adhering to this protocol is optional, it is not clear to what extent developers use it in practice.

A callback is a function that is passed as an argument to another function, which is expected to invoke it either immediately or at some point in the future. Callbacks can be seen as a form of the continuation-passing style (CPS), in which control is passed explicitly in the form of a continuation; in this case the callback passed as an argument represents a continuation.

**Synchronous and asynchronous callbacks**

There are two types of callbacks. A callback passed to a function*f* can be invoked *synchronously*before *f*returns, or it can be deferred to execute *asynchronously* some time after *f*returns.

Diagram

Description automatically generated

Figure 1. The JavaScript event loop model

JavaScript uses an event-driven model with a single thread of execution. Programming with callbacks is especially useful when a caller does not want to wait until the caller completes. To this end, the desired non-blocking operation is scheduled as a callback and the main thread continues its synchronous execution. When the operation completes, a message is enqueued into a task queue along with the provided callback. The event loop in JavaScript prioritizes the single thread to execute the call stack first; when the stack is empty, the event loop dequeues a message from the task queue and executes the corresponding callback function. Figure 1 illustrates the event loop model of JavaScript for a non-blocking HTTP *get*call with a callback named *cb.*

**Named and anonymous callbacks**

JavaScript callbacks can be *named*functions or *anonymous* functions (e.g., lines 2, 3, or 5 of Listing 1). Each approach has its tradeoffs. Named callbacks can be reused and are easily identified in stack traces or breakpoints during debugging activities. However naming causes the callback function to persist in memory and prevents it from being garbage collected. On the other hand, anonymous callbacks are more resource-friendly because they are marked for garbage collection immediately after being executed. However, anonymous callbacks are not reusable and may be difficult to maintain, test, or debug.

**Nested callbacks**

Developers often need to combine several callback-accepting functions together to achieve a certain task. For example, two callbacks have to be nested if the result of the first callback needs to be passed into the second callback in a non-blocking way (see lines 2–8 in Listing 3). This structure becomes more complex when the callbacks need to be conditionally nested. Control-flow composition with nested callbacks increases the complexity of the code. The term “callback hell” has been coined by developers to voice their common frustration with this complexity.

**Error-first callbacks**

In a synchronous JavaScript code the *throw* keyword can be used to signal an error and *try/catch* can be used to handle the error. When there is a asynchrony, however, it may not be possible to handle an error in the context it is thrown. Instead, the error must be propagated asynchronously to an error handler in a different context. Callbacks are the basic mechanism for delivering errors asynchronously in JavaScript. Because there is no explicit language support for asynchronous error signaling, the developer community has proposed a convention — dedicate the first argument in the callback to be a permanent place-holder for error signaling.

More exactly, the *error-first callback* protocol specifies that during a callback invocation, either the first error argument is non-null (indicating an error), or the error argument is null and the other arguments contain data (indicating success), but not both. Listing 2 shows an example of this protocol. If an error occurs while reading the file (line 4), the anonymous function will be called with error as the first argument. For this error to be handled, it will be propagated by passing it as the first argument. For this error to be handled, it will be propagated by passing it as the first argument of the callback (line 5). The error can then be handled at a more appropriate location (lines 17–21). When there is no error, the program continues (line 8) and invokes the callback with the first (error) argument set to null.

var fs = require('fs');  
// read a file  
function read\_the\_file(filename, callback) {  
 fs.readFile(filename, (err, contents) => {  
 if (err) return callback(err);// if no error, continue  
 read\_data\_from\_db(null, contents, callback);  
 });  
}function read\_data\_from\_db(err, contents, callback) {  
// some long running task  
}read\_the\_file('some/file', (err, result) => {  
 if (err) {  
// handle the error  
 console.log(err);  
 return;  
 }  
//do something with the result  
});// Listing 2

The error-first callback protocol is intended to simplify exception handling for developers; if there is an error, it will always propagate as the first argument through the API, and API clients can always check the first argument for errors. But, there is no automatic checking of the error-first protocol in JavaScript. It is therefore unclear how frequently developers adhere to this protocol in practice.

**Handling callbacks**

A quick search in the NPM repository reveals that there are over 4.500 modules to help developers with asynchronous JavaScript programming. Two solutions that are gaining traction in helping developers handle callbacks are libraries, such as *Async.js* and new language features such as *Promises*. We now briefly detail these two approaches.

The Async.js library exposes an API to help developers manage callbacks. For example, Listing 3 shows how nested callbacks in vanilla Javascript (lines 1- 8) can be expressed using the *waterfall* method available in Async.js (11–18)

// Before: nested callbacks  
$("#button").click(() => {  
 promptUserForTwitterHandle((handle) => {  
 twitter.getTweetsFor(handle, (tweets) => {  
 ui.show(tweets);  
 });  
 });  
});// After: Using Async.js waterfall method  
$("#button").click(() => {  
 async.waterfall([  
 promptUserForTwitterHandle,  
 twitter.getTweetsFor,  
 ui.show  
 ],  
 handleError);  
});// After: sequential join of callbacks with Promises  
$("#button").clickPromise()  
 .then(promptUserForTwitterHandle)  
 .then(twitter.getTweetsFor)  
 .then(ui.show);// Listing 3

Promises are a JavaScript language extension. A promise-based function takes some input and returns a promise object representing the result of an asynchronous operation. A promise object can be queried by the developer to answer questions like “were there any errors while executing the async call?” or “has the data from the async call arrived yet?” A promise object, once fulfilled, can notify any function that depends on its data. Listing 3 illustrates how nested callbacks in vanilla JavaScript (lines 1- 8) can be re-expressed using Promises (20–24)

References:

1. Stack Overflow: Developer Survey.
2. G. J. Sussman and G. L Steele Jr. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*
3. M. Ogden: Callback Hell
4. Error Handling in Node.js
5. C. McMahon: Async.js
6. Promises / A+ Promise Specification