8 Essential Node.js Interview Questions *

Consider the following JavaScript code:

```
console.log("first");
setTimeout(function() {
    console.log("second");
}, 0);
console.log("third");
```

The output will be:

```
first
third
second
```

Assuming that this is the desired behavior, and that we are using Node.js version 0.10 or higher, how else might we write this code?

View the answer →Hide answer

Node.js version 0.10 introduced `setImmediate`, which is equivalent to `setTimeout(fn, 0)`, but with some slight advantages.

`setTimeout(fn, delay)` calls the given callback `fn` after the given `delay` has ellapsed (in milliseconds). However, the callback is not executed immediately at this time, but added to the function queue so that it is executed **as soon as possible**, after all the currently executing and currently queued event handlers have completed. Setting the delay to 0 adds the callback to the queue immediately so that it is executed as soon as all currently-queued functions are finished.

`setImmediate(fn)` achieves the same effect, except that it doesn't use the queue of functions. Instead, it checks the queue of I/O event handlers. If all I/O events in the current snapshot are processed, it executes the callback. It queues them immediately after the last I/O handler somewhat like `process.nextTick`. This is faster than `setTimeout(fn, 0)`.

So, the above code can be written in Node as:

```
console.log("first");
setImmediate(function(){
    console.log("second");
});
console.log("third");
```

What is "callback hell" and how can it be avoided?

View the answer →Hide answer

"Callback hell" refers to heavily nested callbacks that have become unweildy or unreadable.

An example of heavily nested code is below:

```
query("SELECT clientId FROM clients WHERE clientName='picanteverde';", function(id){
  query("SELECT * FROM transactions WHERE clientId=" + id, function(transactions){
    transactions.each(function(transac){
      query("UPDATE transactions SET value = " + (transac.value*0.1) + " WHERE id=" + transac.id, function(error){
        if(!error){
          console.log("success!!");
        }else{
          console.log("error");
        }
      });
    });
  });
});
```

The primary method to fix callback hell is usually referred to as **modularization**. The callbacks are broken out into independent functions which can be called with some parameters. So the first level of improvement might be:

```
var logError = function(error){
    if(!error){
      console.log("success!!");
    }else{
      console.log("error");
    }
  },
  updateTransaction = function(t){
    query("UPDATE transactions SET value = " + (t.value*0.1) + " WHERE id=" + t.id, logError);
  },
  handleTransactions = function(transactions){
    transactions.each(updateTransaction);
  },
  handleClient = function(id){
    query("SELECT * FROM transactions WHERE clientId=" + id, handleTransactions);
  };

query("SELECT clientId FROM clients WHERE clientName='picanteverde';",handleClient);
```

Even though this code is much easier to read, and we created some functions that we can even reuse later, in some cases it may be appropriate to use a more robu
solution in the form of **promises**. Promises allow additional desirable behavior such as error propagation and chaining. Node.js doesn't include much core suppo

promises, so one of the popular promise libraries should be used. One of the most popular is the Q promise library.

More information about promises and how they work can be found here.

Additionally, a more supercharged solution to callback hell is provided by **generators**, as these can resolve execution dependency between different callbacks. However, generators are much more advanced and it might be overkill to use them for this purpose. To read more about generators you can start with this post.

How does Node.js handle child threads?

View the answer →Hide answer

Node.js, in its essence, is a **single thread** process. It does not expose child threads and thread management methods to the developer. Technically, Node.js *does* spawn child threads for certain tasks such as asynchronous I/O, but these run behind the scenes and do not execute any application JavaScript code, nor block the main event loop.

If threading support is desired in a Node.js application, there are tools available to enable it, such as the ChildProcess module.

What is the preferred method of resolving unhandled exceptions in Node.js?

View the answer →Hide answer

Unhandled exceptions in Node.js can be caught at the `Process` level by attaching a handler for `uncaughtException` event.

```
process.on('uncaughtException', function(err) {
  console.log('Caught exception: ' + err);
});
```

However, `uncaughtException` is a very crude mechanism for exception handling and may be removed from Node.js in the future. An exception that has bubbled all the way up to the `Process` level means that your application, and Node.js may be in an undefined state, and the only sensible approach would be to restart everything.

The preferred way is to add another layer between your application and the Node.js process which is called the domain.

Domains provide a way to handle multiple different I/O operations as a single group. So, by having your application, or part of it, running in a separate domain, you can safely handle exceptions at the domain level, before they reach the `Process` level.

How does Node.js support multi-processor platforms, and does it fully utilize all processor resources?

View the answer →Hide answer

Since Node.js is by default a **single thread** application, it will run on a single processor core and will not take full advantage of multiple core resources. However, Node.js provides support for deployment on multiple-core systems, to take greater advantage of the hardware. The Cluster module is one of the core Node.js modules and it allows running multiple Node.js worker processes that will share the same port.

What is typically the first argument passed to a Node.js callback handler?

View the answer →Hide answer

Node.js core modules, as well as most of the community-published ones, follow a pattern whereby the first argument to any callback handler is an optional error object. If there is no error, the argument will be null or undefined.

A typical callback handler could therefore perform error handling as follows:

```
function callback(err, results) {
    // usually we'll check for the error before handling results
    if(err) {
        // handle error somehow and return
    }
    // no error, perform standard callback handling
}
```

Consider following code snippet:

```
{
    console.time("loop");
    for (var i = 0; i < 1000000; i += 1){
        // Do nothing
    }
    console.timeEnd("loop");
}
```

The time required to run this code in Google Chrome is considerably more than the time required to run it in Node.js. Explain why this is so, even though both use the v8 JavaScript Engine.

View the answer →Hide answer

Within a web browser such as Chrome, declaring the variable `i` outside of any function's scope makes it global and therefore binds it as a property of the `window` object. As a result, running this code in a web browser requires repeatedly resolving the property `i` within the heavily populated `window` namespace in each iteration of the `for` loop.

In Node.js, however, declaring any variable outside of any function's scope binds it only to the module's own scope (not the `window` object) which therefore makes it much easier and faster to resolve.

What is REPL? What purpose it is used for?

View the answer →Hide answer

REPL stands for (READ, EVAL, PRINT, LOOP). Node js comes with bundled REPL environment. This allows for the easy creation of CLI (Command Line Interface) applications.

* There is more to interviewing than tricky technical questions, so these are intended merely as a guide. Not every “A” candidate worth hiring will be able to answer them all, nor does answering them all guarantee an “A” candidate. At the end of the day, hiring remains an art, a science — and a lot of work.
Submit an interview question
Submitted questions and answers are subject to review and editing, and may or may not be selected for posting, at the sole discretion of Toptal, LLC.

Name
Email
Enter your question here
Enter your answer here
All fields are required
I agree with the Terms and Conditions of Toptal, LLC's Privacy Policy
Submit a Question

Thanks for submitting your question.
Our editorial staff will review it shortly. Please note that submitted questions and answers are subject to review and editing, and may or may not be selected for posting, at the sole discretion of Toptal, LLC.
Looking for Node.js experts? Check out Toptal’s Node.js developers.
View full profile »
Nam Nguyen
United States
Nam is a full-stack web developer with proficiency in both Node.js and .NET. Interviewers describe him as punctual and pleasant with excellent programming instincts. He strongly believes that a complete code should not just work, but also be clean and maintainable.
Node.jsjQuery
Hire Nam
View full profile »
Alejandro Hernandez
Argentina
Alejandro got his bachelor's degree in software engineering in 2005 and has since been working for software companies of all sizes from all around the globe as a freelancer. Currently, he enjoys working as a full-stack architect in JavaScript projects, where his experience and his deep understanding of architecture and theory are most impactful.
Node.jsJavaScriptReact
Hire Alejandro
View full profile »
Daniel Lauzon
Canada
Daniel is a a technology enthusiast and a very proficient programmer. He holds a Ph.D. in Information Theory, and an M.Sc. in Mathematics. He has also built a successful enterprise software company.
Node.jsPHPGroovyJavaScript+9 more
Hire Daniel
Toptal connects the top 3% of freelance talent all over the world.

Join the Toptal community.

Hire a developer
or
Apply as a developer

Highest In-Demand Talent

- iOS Developers
- Front-End Developers
- UX Designers
- UI Designers
- Financial Modeling Consultants
- Interim CFOs