Node.js Interview Questions

Q1. What is Node.js? What is it used for?

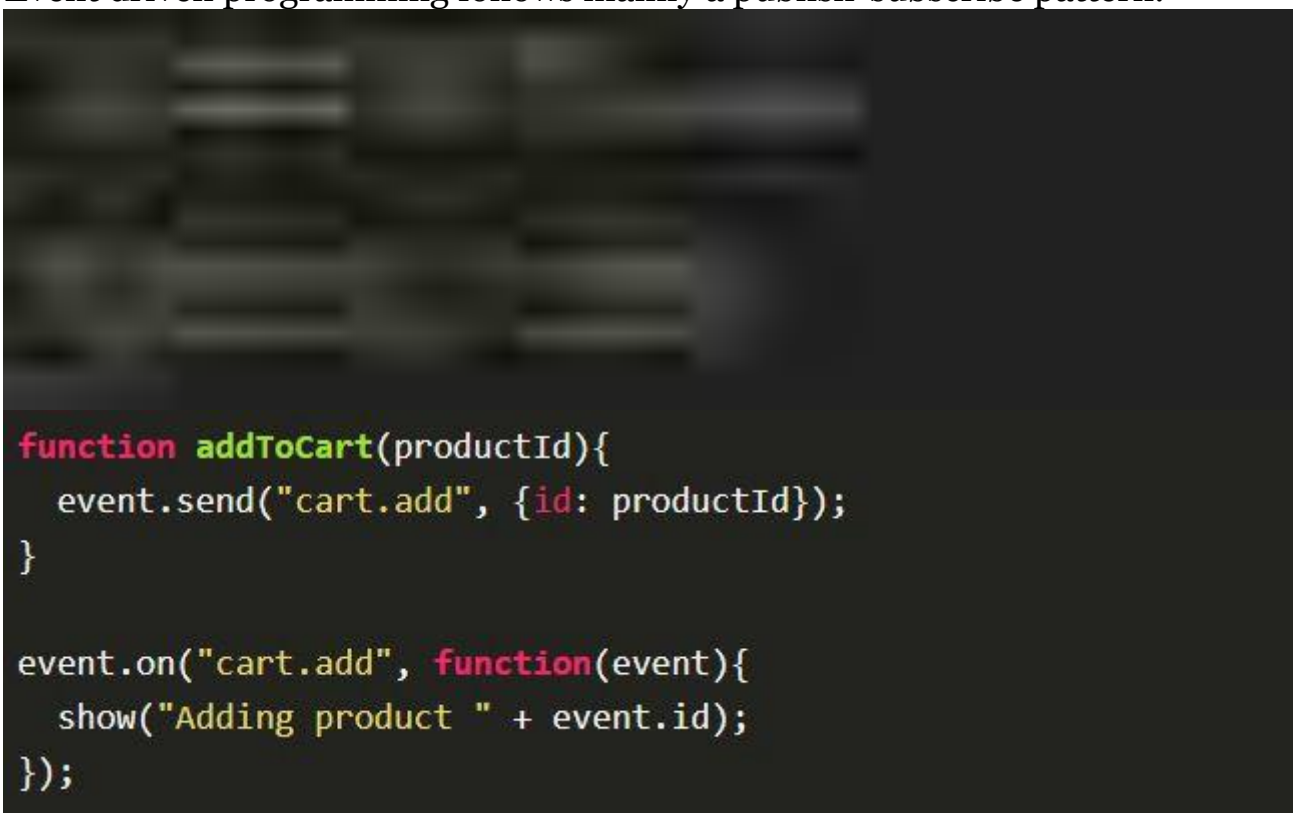
Node.js is a run-time JavaScript environment built on top of Chrome's V8 engine. It uses an event-driven, non-blocking I/O model. It is lightweight and so efficient. Node.js has a package ecosystem called **npm**.

Node.js can be used to build different types of applications such as web application, real-time chat application, REST API server etc. However, it is mainly used to build network programs like web servers, similar to PHP, Java, or ASP.NET. Node.js was developed by Ryan Dahl in 2009.

Q2. What is Event-driven programming?

Event-driven programming is building our application based on and respond to events. When an event occurs, like click or keypress, we are running a callback function which is registered to the element for that event.

Event driven programming follows mainly a publish-subscribe pattern.



```
function addToCart(productId){
  event.send("cart.add", {id: productId});
}

event.on("cart.add", function(event){
  show("Adding product " + event.id);
});
```

Q3. What is *Event loop* in Node.js work? And How does it work?

The *Event loop* handles all async callbacks. Node.js (or JavaScript) is a single-threaded, event-driven language. This means that we can attach listeners to events, and when a said event fires, the listener executes the callback we provided.

Whenever we are call `setTimeout`, `http.get` and `fs.readFile`, Node.js runs this operations and further continue to run other code without waiting for the output. When the operation is finished, it receives the output and runs our callback function.

So all the callback functions are queued in an loop, and will run one-by-one when the response has been received.

Q4. What is REPL in Node.js?

REPL means Read-Eval-Print-Loop. It is a virtual environment that comes with Node.js. We can quickly test our JavaScript code in the Node.js REPL environment.

To launch the REPL in Node.js, just open the command prompt and type `node`. It will change the prompt to `>` in Windows and MAC.

Now we can type and run our JavaScript easily. For example, if we type `10 + 20`, it will print `30` in the next line.

Q5. What is the purpose of module.exports in Node.js?

A module encapsulates related code into a single unit of code. This can be interpreted as moving all related functions into a file. Imagine that we created a file called `greetings.js` and it contains the following two functions:

```
module.exports = {
  sayHelloInEnglish: function() {
    return "HELLO";
  },

  sayHelloInSpanish: function() {
    return "Hola";
  }
};
```

In the above code, `module.exports` exposes two functions to the outer world. We can import them in another file as follow:

```
var greetings = require("./greetings.js");

greetings.sayHelloInEnglish(); // Hello

greetings.sayHelloInSpanish(); //Hola
```

Q6. What is the difference between Asynchronous and Non-blocking?

Asynchronous literally means not synchronous. We are making HTTP requests which are asynchronous, means we are not waiting for the server response. We continue with other block and respond to the server response when we received.

The term Non-Blocking is widely used with IO. For example non-blocking read/write calls return with whatever they can do and expect caller to execute the call again. Read will wait until it has some data and put calling thread to sleep.

Q7. What is Tracing in Node.js?

Tracing provides a mechanism to collect tracing information generated by V8, Node core and userspace code in a log file. Tracing can be enabled by passing the `--trace-events-enabled` flag when starting a Node.js application.

```
node --trace-events-enabled --trace-event-categories v8,node server.js
```

The set of categories for which traces are recorded can be specified using the `--trace-event-categories` flag followed by a list of comma separated category names. By default the `node` and `v8` categories are enabled.

Running Node.js with tracing enabled will produce log files that can be opened in the `chrome://tracing` tab of Chrome.

Q8. How will you debug an application in Node.js?

Node.js includes a debugging utility called `debugger`. To enable it start the Node.js with the `debug` argument followed by the path to the script to debug.

Inserting the statement `debugger;` into the source code of a script will enable a breakpoint at that position in the code:

```
x = 5;
setTimeout(() => {
  debugger;
  console.log('world');
}, 1000);
```

Q9. Difference between setImmediate() vs setTimeout()

`setImmediate()` and `setTimeout()` are similar, but behave in different ways depending on when they are called.

- `setImmediate()` is designed to execute a script once the current poll (event loop) phase completes.
- `setTimeout()` schedules a script to be run after a minimum threshold in ms has elapsed.

The order in which the timers are executed will vary depending on the context in which they are called. If both are called from within the main module, then timing will be bound by the performance of the process.

Q10. What is process.nextTick()

`setImmediate()` and `setTimeout()` are based on the event loop.
But `process.nextTick()` technically not part of the event loop. Instead, the `nextTickQueue` will be processed after the current operation completes, regardless of the current phase of the event loop.

Thus, any time you call process.nextTick() in a given phase, all callbacks passed to process.nextTick() will be resolved before the event loop continues.

Q11. What is package.json? What is it used for?

This file holds various metadata information about the project. This file is used to give information to `npm` that allows it to identify the project as well as handle the project's dependencies.

Some of the fields are: `name`, `name`, `description`, `author` and `dependencies`.

When someone installs our project through `npm`, all the dependencies listed will be installed as well. Additionally, if someone runs `npm install` in the root directory of our project, it will install all the dependencies to `./node_modules` directory.

Q12. What is libuv?

`libuv` is a multi-platform support library with a focus on asynchronous I/O. It was primarily developed for use by Node.js, but it's also used by Luvit, Julia, pyuv, and others.

When the node.js project began in 2009 as a JavaScript environment decoupled from the browser, it is using Google's V8 and Marc Lehmann's `libev`, node.js

combined a model of I/O – evented – with a language that was well suited to the style of programming; due to the way it had been shaped by browsers. As node.js grew in popularity, it was important to make it work on Windows, but libev ran only on Unix. `libuv` was an abstraction around libev or IOCP depending on the platform, providing users an API based on libev. In the node-v0.9.0 version of libuv libev was removed.

Some of the features of `libuv` are:

- Full-featured event loop backed by epoll, kqueue, IOCP, event ports.
- Asynchronous TCP and UDP sockets
- Asynchronous file and file system operations
- Child processes
- File system events

Q13. What are some of the most popular modules of Node.js?

There are many most popular, most starred or most downloaded modules in Node.js. Some of them are:

- express
- async
- browserify
- socket.io
- bower
- gulp
- grunt

Q14. What is EventEmitter in Node.js?

All objects that emit events are instances of the `EventEmitter` class. These objects expose an `eventEmitter.on()` function that allows one or more functions to be attached to named events emitted by the object.

When the `EventEmitter` object emits an event, all of the functions attached to that specific event are called *synchronously*.

```
const events = require('events');
const eventEmitter = new events.EventEmitter();

let myEvent = function ringBell() {
  console.log('Event is emitted');
}

eventEmitter.on('emitEvent', myEvent);

eventEmitter.emit('emitEvent');
```

Q15. What is Streams in Node.js?

Streams are pipes that let you easily read data from a source and pipe it to a destination. Simply put, a stream is nothing but an `EventEmitter` and implements some specials methods. Depending on the methods implemented, a stream becomes Readable, Writable, or Duplex (both readable and writable).

For example, if we want to read data from a file, the best way to do it from a stream is to listen to data event and attach a callback. When a chunk of data is available, the readable stream emits a data event and your callback executes. Take a look at the following snippet:

```
var fs = require('fs');
var readableStream = fs.createReadStream('textFile.txt');
var fileData = '';

readableStream.on('data', function(chunk) {
  data += chunk;
});

readableStream.on('end', function() {
  console.log(data);
});
```

Types of streams are: Readable, Writable, Duplex and Transform.

Q16. What is the difference between readFile vs createReadStream in Node.js?

readFile — is for asynchronously reads the entire contents of a file. It will read the file completely into memory before making it available to the User. `readFileSync` is synchronous version of `readFile`.

createReadStream — It will read the file in chunks of the default size 64 kb which is specified before hand.

Q17. What is crypto in Node.js? How do you cipher the secured information in Node.js?

The `crypto` module in Node.js provides cryptographic functionality that includes a set of wrappers for OpenSSL's hash, HMAC, cipher, decipher, sign and verify functions.

```
var crypto = require('crypto');

var secret = 'abcdefg';
var hash = crypto.createHmac('sha256', secret)
  .update('I love cupcakes')
  .digest('hex');
console.log(hash);

  //    c0fa1bc00531bd78ef38c628449c5102aeabd49b5dc3a2a516ea6ea959d6658e
```

Q18. What is the use of Timers is Node.js?

The Timers module in Node.js contains functions that execute code after a set period of time. Timers do not need to be imported via `require()`, since all the methods are available globally to emulate the browser JavaScript API.

The Node.js API provides several ways of scheduling code to execute at some point after the present moment. The functions below may seem familiar, since they are available in most browsers, but Node.js actually provides its own implementation of these methods.

Node.js Timer provides `setTimeout()`, `setImmediate()` and `setInterval`.

Q19. What is the use of DNS module in Node.js?

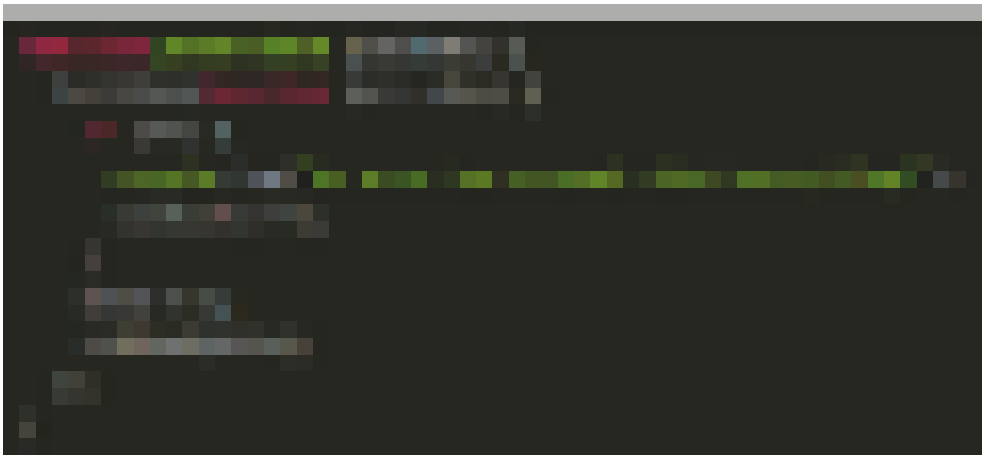
`dns` module which provide underlying system's name resolution and DNS look up facilities. DNS module consists of an asynchronous network wrapper.

The most commonly used functions in DNS module are:

- **dns.lookup(adress, options, callback)** - The dns lookup method takes any website address as its first parameter and returns the corresponding first IPV4 or IPV6 record. The options parameter can be an integer or object. If no options are provided both IPV4 and IPV6 are valid inputs. The third parameter is the callback functions.
- **dns.lookupservice(address, port, callback)** - This function converts any physical address such as "www.knowledgehills.com" to array of record types. The record types are specified by the second parameter "rrbyte". Finally the third method is the callback function.
- **dns.getServers()** - This function returns an array of IP address strings, formatted according to rfc5952, that are currently configured for DNS resolution. A string will include a port section if a custom port is used.
- **dns.setServers()** - This function sets the IP address and port of servers to be used when performing DNS resolution. The `dns.setServers()` method must not be called while a DNS query is in progress.

Q20. What is a Callback function in Node.js?

Node.js, being an asynchronous platform, doesn't wait around for things like file I/O to finish — Node.js uses callbacks. A callback is a function called at the completion of a given task; this prevents any blocking, and allows other code to be run in the meantime.



```
function processData (callback) {
  fetchData(function (err, data) {
    if (err) {
      console.log("An error has occured. Abort everything!");
      callback(err);
    }
    data += 1;
    callback(data);
  });
}
```

Callbacks are the foundation of Node.js. Callbacks give us an interface with which to say, “and when you’re done doing that, do all this.” This allows us to have as many IO operations as our OS can handle happening at the same time. For example, in a web server with hundreds or thousands of pending requests with multiple blocking queries, performing the blocking queries asynchronously gives you the ability to be able to continue working and not just sit still and wait until the blocking operations come back.

Q21. What are the security mechanisms available in Node.js?

We can secure our Node.js application in the following ways:

Authentication — Authentication is one of the primary security stages at which user is identified as permitted to access the application at all. Authentication verifies the user’s identity through one or several checks. In Node.js, authentication can be either session-based or token-based. In session-based authentication, the user’s credentials are compared to the user account stored on the server and, in the event of successful validation, a session is started for the user. Whenever the session expires, the user needs to log in again. In token-based authentication, the user’s credentials are applied to generate a string called a token which is then associated with the user’s requests to the server.

Error Handling — Usually, the error message contains the explanation of what’s actually gone wrong for the user to understand the reason. At the same time, when the error is related to the application code syntax, it can be set to display the entire log content on the frontend. For an experienced hacker, the log content can reveal a lot of sensitive internal information about the application code structure and tools used within the software.

Request Validation — Another aspect which has to be considered, while building a secure Node.js application, is a validation of requests or, in other

words, a check of the incoming data for possible inconsistencies. It may seem that invalid requests do not directly affect the security of a Node.js application, however, they may influence its performance and robustness. Validating the incoming data types and formats and rejecting requests not conforming to the set rules can be an additional measure of securing your Node.js application.

Node.js Security Tools and Best Practices — We can use tools like **helmet** (protects our application by setting HTTP headers), **csurf** (validates tokens in incoming requests and rejects the invalid ones), **node rate limiter** (controls the rate of repeated requests. This function can protect you from brute force attacks) and **cors** (enables cross-origin resource sharing).

Q22. What is the passport in Node.js?

Passport.js is a simple, unobtrusive Node.js authentication middleware for Node.js. Passport.js can be dropped into any Express.js-based web application.

Passport recognizes that each application has unique authentication requirements. Authentication mechanisms, known as strategies, are packaged as individual modules. Applications can choose which strategies to employ, without creating unnecessary dependencies.

By default, if authentication fails, Passport will respond with a `401 Unauthorized` status, and any additional route handlers will not be invoked. If authentication succeeds, the `next` handler will be invoked and the req.user property will be set to the authenticated user.

Node.js Interview Questions

- What is an error-first callback?
- How can you avoid callback hells?
- What are Promises?
- What tools can be used to assure consistent style? Why is it important?
- When should you npm and when yarn?
- What's a stub? Name a use case!
- What's a test pyramid? Give an example!
- What's your favorite HTTP framework and why?
- How can you secure your HTTP cookies against XSS attacks?
- How can you make sure your dependencies are safe?

The Answers

What is an error-first callback?

Error-first callbacks are used to pass errors and data as well. You have to pass the error as the first parameter, and it has to be checked to see if something went wrong. Additional arguments are used to pass data.

```
fs.readFile(filePath, function(err, data) {
 if (err) {
  // handle the error, the return is important here
  // so execution stops here
  return console.log(err)
 }
 // use the data object
 console.log(data)
})
```

How can you avoid callback hells?

There are lots of ways to solve the issue of callback hells:

- **modularization**: break callbacks into independent functions
- use a **control flow library**, like async
- use **generators with Promises**
- use **async/await** *(note that it is only available in the latest v7 release and not in the LTS version - you can read our experimental async/await how-to here)*

Q: How to avoid callback hells? A: modularization, control flow libraries, generators with promises, async/await

What are Promises?

Promises are a concurrency primitive, first described in the 80s. Now they are part of most modern programming languages to make your life easier. Promises can help you better handle async operations.

An example can be the following snippet, which after 100ms prints out the `result` string to the standard output. Also, note the `catch`, which can be used for error handling. Promises are chainable.

```
new Promise((resolve, reject) => {
```

```
setTimeout(() => {
  resolve('result')
}, 100)
})
.then(console.log)
.catch(console.error)
```

What tools can be used to assure consistent style? Why is it important?

When working in a team, consistent style is important, so team members can modify more projects easily, without having to get used to a new style each time.

Also, it can help eliminate programming issues using static analysis.

Tools that can help:

- ESLint
- Standard

If you'd like to be even more confident, I suggest you to learn and embrace the JavaScript Clean Coding principles as well!

What's a stub? Name a use case!

Stubs are functions/programs that simulate the behaviors of components/modules. Stubs provide canned answers to function calls made during test cases.

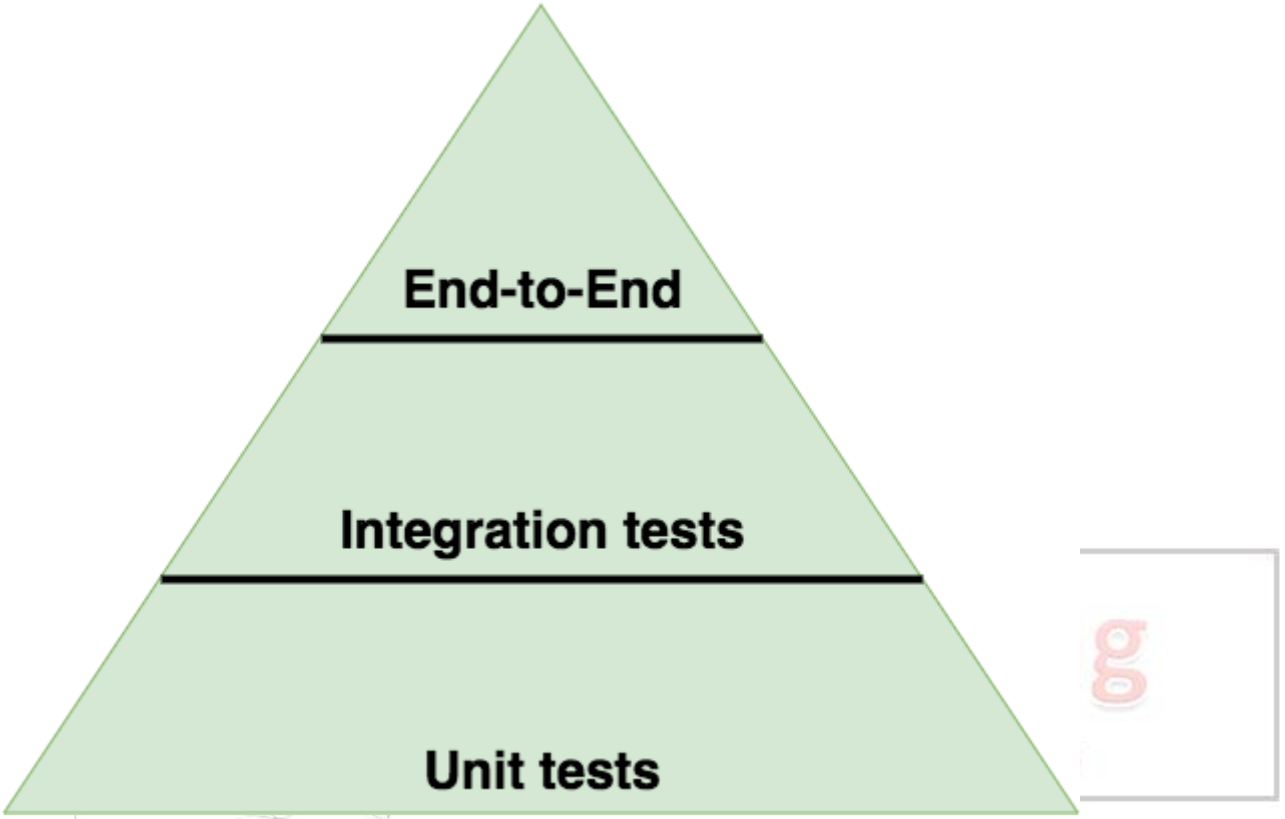
An example can be writing a file, without actually doing so.

```
var fs = require('fs')

var writeFileStub = sinon.stub(fs, 'writeFile', function (path, data, cb) {
  return cb(null)
})

expect(writeFileStub).to.be.called
```

```
writeFileStub.restore()
```

What's a test pyramid? Give an example!

A test pyramid describes the ratio of how many unit tests, integration tests and end-to-end test you should write.



An example for an HTTP API may look like this:

- lots of low-level unit tests for models *(dependencies **are stubbed**)*,
- fewer integration tests, where you check how your models interact with each other *(dependencies **are not stubbed**)*,
- less end-to-end tests, where you call your actual endpoints *(dependencies **are not stubbed**)*.

What's your favorite HTTP framework and why?

There is no right answer for this. The goal here is to understand how deeply one knows the framework she/he uses. Tell what are the pros and cons of picking that framework.

When are background/worker processes useful? How can you handle worker tasks?

Worker processes are extremely useful if you'd like to do data processing in the background, like sending out emails or processing images.

There are lots of options for this like RabbitMQ or Kafka.

How can you secure your HTTP cookies against XSS attacks?

XSS occurs when the attacker injects executable JavaScript code into the HTML response.

To mitigate these attacks, you have to set flags on the `set-cookie` HTTP header:

- **HttpOnly** - this attribute is used to help prevent attacks such as cross-site scripting since it does not allow the cookie to be accessed via JavaScript.
- **secure** - this attribute tells the browser to only send the cookie if the request is being sent over HTTPS.

So it would look something like this: `Set-Cookie: sid=<cookie-value>; HttpOnly`. If you are using Express, with express-cookie session, it is working by default.

How can you make sure your dependencies are safe?

When writing Node.js applications, **ending up with hundreds or even thousands of dependencies can easily happen**.
For example, if you depend on Express, you depend on 27 other modules directly, and of course on those dependencies' as well, so manually checking all of them is not an option!

The only option is to automate the update / security audit of your dependencies. For that there are free and paid options:

- `npm outdated`
- Trace by RisingStack
- NSP
- GreenKeeper
- Snyk

Node.js Interview Puzzles

The following part of the article is useful if you'd like to prepare for an interview that involves puzzles, or tricky questions.

What's wrong with the code snippet?

```
new Promise((resolve, reject) => {
  throw new Error('error')
}).then(console.log)
```

The Solution

As there is no `catch` after the `then`. This way the error will be a silent one, there will be no indication of an error thrown.

To fix it, you can do the following:

```
new Promise((resolve, reject) => {
  throw new Error('error')
}).then(console.log).catch(console.error)
```

If you have to debug a huge codebase, and you don't know which Promise can potentially hide an issue, you can use the `unhandledRejection` hook. It will print out all unhandled Promise rejections.

```
process.on('unhandledRejection', (err) => {
  console.log(err)
})
```

What's wrong with the following code snippet?

```
function checkApiKey (apiKeyFromDb, apiKeyReceived) {
  if (apiKeyFromDb === apiKeyReceived) {
    return true
  }
  return false
}
```

The Solution

When you compare security credentials it is crucial that you don't leak any information, so you have to make sure that you compare them in fixed time. If you fail to do so, your application will be vulnerable to timing attacks.

But why does it work like that?

V8, the JavaScript engine used by Node.js, tries to optimize the code you run from a performance point of view. It starts comparing the strings character by character, and once a mismatch is found, it stops the comparison operation. **So the longer the attacker has right from the password, the more time it takes.**

To solve this issue, you can use the npm module called cryptiles.

```
function checkApiKey (apiKeyFromDb, apiKeyReceived) {
  return cryptiles.fixedTimeComparison(apiKeyFromDb, apiKeyReceived)
}
```

What's the output of following code snippet?

```
Promise.resolve(1)
  .then((x) => x + 1)
  .then((x) => { throw new Error('My Error') })
  .catch(() => 1)
  .then((x) => x + 1)
  .then((x) => console.log(x))
  .catch(console.error)
```

The Answer

The short answer is `2` - however with this question **I'd recommend asking the candidates to explain what will happen line-by-line to understand how they think**. It should be something like this:

1. A new Promise is created, that will resolve to `1`.
2. The resolved value is incremented with 1 (so it is `2` now), and returned instantly.
3. The resolved value is discarded, and an error is thrown.

4. The error is discarded, and a new value (`1`) is returned.
5. The execution did not stop after the catch, but before the exception was handled, it continued, and a new, incremented value (`2`) is returned.
6. The value is printed to the standard output.
7. This line won't run, as there was no exception.