# Node Js QA

**What's wrong with the code snippet?**

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

**throw new Error('error')**

**}).then(console.log)**

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.

Fix :

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

**What's wrong with the following code snippet?**

**function** **checkApiKey** (apiKeyFromDb, apiKeyReceived) {

**if** (apiKeyFromDb === apiKeyReceived) {

**return** true

}

**return** false

}

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](https://en.wikipedia.org/wiki/Timing_attack).

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](https://www.npmjs.com/package/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 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:

A new Promise is created, that will resolve to 1.

The resolved value is incremented with 1 (so it is 2 now), and returned instantly.

The resolved value is discarded, and an error is thrown.

The error is discarded, and a new value (1) is returned.

The execution did not stop after the catch, but before the exception was handled, it continued, and a new, incremented value (2) is returned.

The value is printed to the standard output.

This line won't run, as there was no exception.

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

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](http://nodejs.org/api/domain.html).

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.

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

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

}

**What will be the output of the program**

**console.log("first");**

**setTimeout(function() {**

**console.log("second");**

**}, 0);**

**console.log("third");**

Output

first

third

second

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");

**Can you name two programming paradigms important for JavaScript app developers?**

JavaScript is a multi-paradigm language, supporting imperative/procedural programming along with OOP (Object-Oriented Programming) and functional programming. JavaScript supports OOP with prototypal inheritance.

Good to hear:

Prototypal inheritance (also: prototypes, OLOO).

Functional programming (also: closures, first class functions, lambdas).

**What is functional programming?**

Functional programming produces programs by composing mathematical functions and avoids shared state & mutable data. Lisp (specified in 1958) was among the first languages to support functional programming, and was heavily inspired by lambda calculus. Lisp and many Lisp family languages are still in common use today.

Functional programming is an essential concept in JavaScript (one of the two pillars of JavaScript). Several common functional utilities were added to JavaScript in ES5.

Good to hear:

Pure functions / function purity.

Avoid side-effects.

Simple function composition.

Examples of functional languages: Lisp, ML, Haskell, Erlang, Clojure, Elm, F Sharp, OCaml, etc…

Mention of features that support FP: first-class functions, higher order functions, functions as arguments/values..

**What is the difference between classical inheritance and prototypal inheritance?**

Class Inheritance: instances inherit from classes (like a blueprint — a description of the class), and create sub-class relationships: hierarchical class taxonomies. Instances are typically instantiated via constructor functions with the `new` keyword. Class inheritance may or may not use the `class` keyword from ES6.

Prototypal Inheritance: instances inherit directly from other objects. Instances are typically instantiated via factory functions or `Object.create()`. Instances may be composed from many different objects, allowing for easy selective inheritance.

In JavaScript, prototypal inheritance is simpler &

more flexible than class inheritance.

Classes: create tight coupling or hierarchies/taxonomies.

Prototypes: mentions of concatenative inheritance, prototype delegation, functional inheritance, object composition.

**What are the pros and cons of functional programming vs object-oriented programming?**

OOP Pros: It’s easy to understand the basic concept of objects and easy to interpret the meaning of method calls. OOP tends to use an imperative style rather than a declarative style, which reads like a straight-forward set of instructions for the computer to follow.

OOP Cons: OOP Typically depends on shared state. Objects and behaviors are typically tacked together on the same entity, which may be accessed at random by any number of functions with non-deterministic order, which may lead to undesirable behavior such as race conditions.

FP Pros: Using the functional paradigm, programmers avoid any shared state or side-effects, which eliminates bugs caused by multiple functions competing for the same resources. With features such as the availability of point-free style (aka tacit programming), functions tend to be radically simplified and easily recomposed for more generally reusable code compared to OOP.

FP also tends to favor declarative and denotational styles, which do not spell out step-by-step instructions for operations, but instead concentrate on what to do, letting the underlying functions take care of the how. This leaves tremendous latitude for refactoring and performance optimization, even allowing you to replace entire algorithms with more efficient ones with very little code change. (e.g., memoize, or use lazy evaluation in place of eager evaluation.)

Computation that makes use of pure functions is also easy to scale across multiple processors, or across distributed computing clusters without fear of threading resource conflicts, race conditions, etc…

FP Cons: Over exploitation of FP features such as point-free style and large compositions can potentially reduce readability because the resulting code is often more abstractly specified, more terse, and less concrete.

More people are familiar with OO and imperative programming than functional programming, so even common idioms in functional programming can be confusing to new team members.

FP has a much steeper learning curve than OOP because the broad popularity of OOP has allowed the language and learning materials of OOP to become more conversational, whereas the language of FP tends to be much more academic and formal. FP concepts are frequently written about using idioms and notations from lambda calculus, algebras, and category theory, all of which requires a prior knowledge foundation in those domains to be understood.

Mentions of trouble with shared state, different things competing for the same resources, etc…

Awareness of FP’s capability to radically simplify many applications.

Awareness of the differences in learning curves.

Articulation of side-effects and how they impact program maintainability.

Awareness that a highly functional codebase can have a steep learning curve.

Awareness that a highly OOP codebase can be extremely resistant to change and very brittle compared to an equivalent FP codebase.

Awareness that immutability gives rise to an extremely accessible and malleable program state history, allowing for the easy addition of features like infinite undo/redo, rewind/replay, time-travel debugging, and so on. Immutability can be achieved in either paradigm, but a proliferation of shared stateful objects complicates the implementation in OOP.

**When is prototypal inheritance an appropriate choice?**

There is more than one type of prototypal inheritance:

Delegation (i.e., the prototype chain).

Concatenative (i.e. mixins, `Object.assign()`).

Functional (Not to be confused with functional programming. A function used to create a closure for private state/encapsulation).

Each type of prototypal inheritance has its own set of use-cases, but all of them are equally useful in their ability to enable composition, which creates has-a or uses-a or can-do relationships as opposed to the is-a relationship created with class inheritance.

In situations where modules or functional programming don’t provide an obvious solution.

When you need to compose objects from multiple sources.

Any time you need inheritance.

**What does “favor object composition over class inheritance” mean?**

This is a quote from “Design Patterns: Elements of Reusable Object-Oriented Software”. It means that code reuse should be achieved by assembling smaller units of functionality into new objects instead of inheriting from classes and creating object taxonomies.

In other words, use can-do, has-a, or uses-a relationships instead of is-a relationships.

Avoid class hierarchies.

Avoid brittle base class problem.

Avoid tight coupling.

Avoid rigid taxonomy (forced is-a relationships that are eventually wrong for new use cases).

Avoid the gorilla banana problem (“what you wanted was a banana, what you got was a gorilla holding the banana, and the entire jungle”).

Make code more flexible.

**What are two-way data binding and one-way data flow, and how are they different?**

Two way data binding means that UI fields are bound to model data dynamically such that when a UI field changes, the model data changes with it and vice-versa.

One way data flow means that the model is the single source of truth. Changes in the UI trigger messages that signal user intent to the model (or “store” in React). Only the model has the access to change the app’s state. The effect is that data always flows in a single direction, which makes it easier to understand.

One way data flows are deterministic, whereas two-way binding can cause side-effects which are harder to follow and understand.

React is the new canonical example of one-way data flow, so mentions of React are a good signal. Cycle.js is another popular implementation of uni-directional data flow.

Angular is a popular framework which uses two-way binding.

**What are the pros and cons of monolithic vs microservice architectures?**

A monolithic architecture means that your app is written as one cohesive unit of code whose components are designed to work together, sharing the same memory space and resources.

A microservice architecture means that your app is made up of lots of smaller, independent applications capable of running in their own memory space and scaling independently from each other across potentially many separate machines.

Monolithic Pros: The major advantage of the monolithic architecture is that most apps typically have a large number of cross-cutting concerns, such as logging, rate limiting, and security features such audit trails and DOS protection.

When everything is running through the same app, it’s easy to hook up components to those cross-cutting concerns.

There can also be performance advantages, since shared-memory access is faster than inter-process communication (IPC).

Monolithic cons: Monolithic app services tend to get tightly coupled and entangled as the application evolves, making it difficult to isolate services for purposes such as independent scaling or code maintainability.

Monolithic architectures are also much harder to understand, because there may be dependencies, side-effects, and magic which are not obvious when you’re looking at a particular service or controller.

Microservice pros: Microservice architectures are typically better organized, since each microservice has a very specific job, and is not concerned with the jobs of other components. Decoupled services are also easier to recompose and reconfigure to serve the purposes of different apps (for example, serving both the web clients and public API).

They can also have performance advantages depending on how they’re organized because it’s possible to isolate hot services and scale them independent of the rest of the app.

Microservice cons: As you’re building a new microservice architecture, you’re likely to discover lots of cross-cutting concerns that you did not anticipate at design time. A monolithic app could establish shared magic helpers or middleware to handle such cross-cutting concerns without much effort.

In a microservice architecture, you’ll either need to incur the overhead of separate modules for each cross-cutting concern, or encapsulate cross-cutting concerns in another service layer that all traffic gets routed through.

Eventually, even monolthic architectures tend to route traffic through an outer service layer for cross-cutting concerns, but with a monolithic architecture, it’s possible to delay the cost of that work until the project is much more mature.

Microservices are frequently deployed on their own virtual machines or containers, causing a proliferation of VM wrangling work. These tasks are frequently automated with container fleet management tools.

Positive attitudes toward microservices, despite the higher initial cost vs monolthic apps. Aware that microservices tend to perform and scale better in the long run.

Practical about microservices vs monolithic apps. Structure the app so that services are independent from each other at the code level, but easy to bundle together as a monolithic app in the beginning. Microservice overhead costs can be delayed until it becomes more practical to pay the price.

**What is asynchronous programming, and why is it important in JavaScript?**

Synchronous programming means that, barring conditionals and function calls, code is executed sequentially from top-to-bottom, blocking on long-running tasks such as network requests and disk I/O.

Asynchronous programming means that the engine runs in an event loop. When a blocking operation is needed, the request is started, and the code keeps running without blocking for the result. When the response is ready, an interrupt is fired, which causes an event handler to be run, where the control flow continues. In this way, a single program thread can handle many concurrent operations.

User interfaces are asynchronous by nature, and spend most of their time waiting for user input to interrupt the event loop and trigger event handlers.

Node is asynchronous by default, meaning that the server works in much the same way, waiting in a loop for a network request, and accepting more incoming requests while the first one is being handled.

This is important in JavaScript, because it is a very natural fit for user interface code, and very beneficial to performance on the server.