

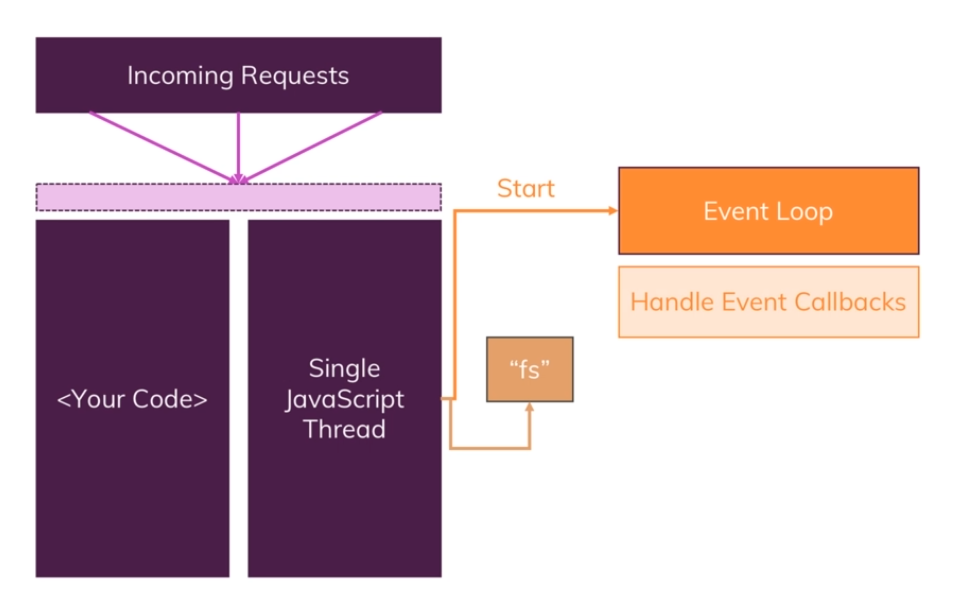
**How Node.js executes code to stay performant and perform time-taking tasks?**

Node.js uses only a single JS thread. (A thread is a part of process in the OS)

If it uses one thread, how is it able to handle multiple incoming requests?

(As if we’re not able to assign a new thread for each request, they ultimately end up all running in the same thread, which poses a security question)

Let's say we have some code which accesses the file system as we already did in this course too. Now working with files often is a task that takes longer because files can be very big and it doesn't necessarily complete instantly, therefore if we're doing this upon an incoming request, a second request might have to wait because we're not able to handle it yet or it even gets declined, so basically our webpage is down for that user.

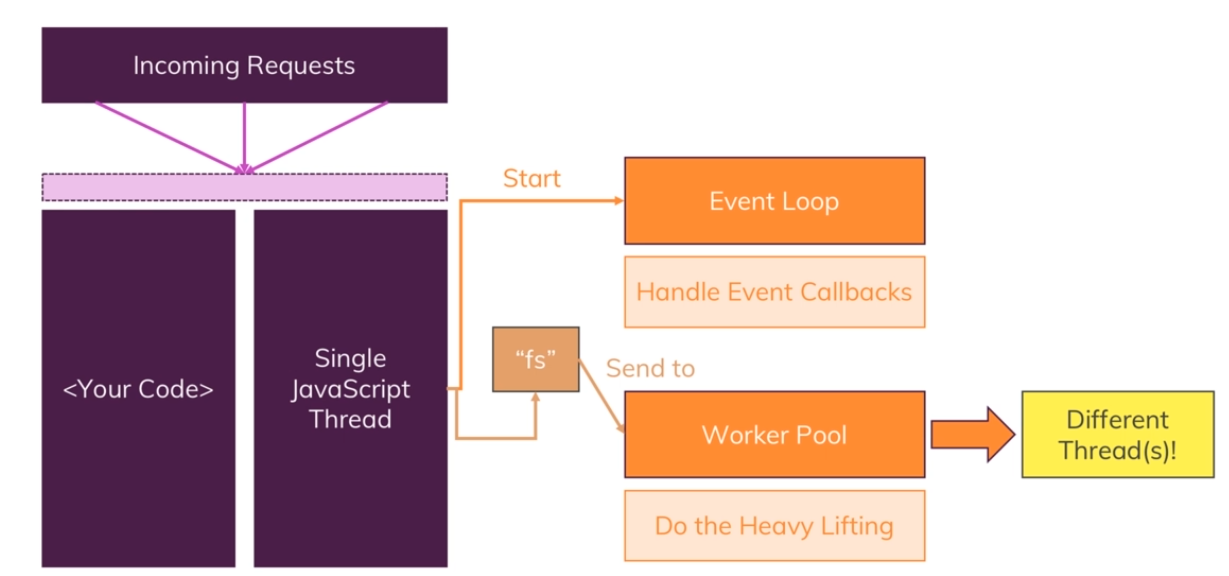


Now one important construct I already mentioned is that event loop, the event loop is automatically started by Node.js when your program starts, you don't have to do that explicitly, Node.js does that when well it basically starts running code.

This is responsible for handling event callbacks though, so all these nice functions we basically added thus far in create server for example, the event loop is responsible for basically running that code when a certain event occurs you could say, it's aware of all these callbacks and execute said code.

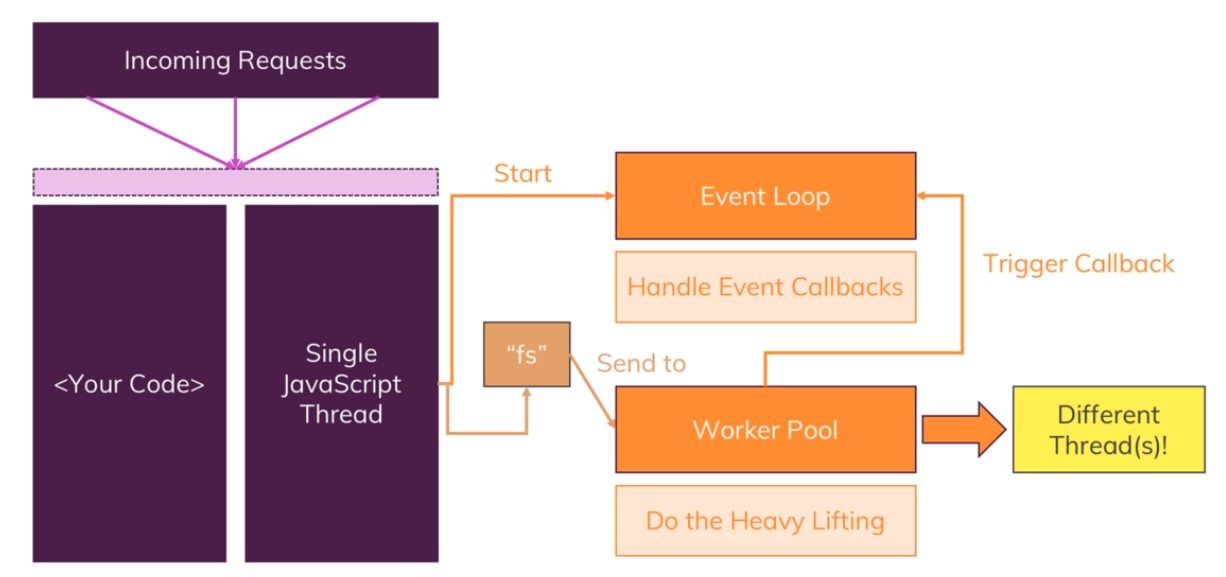
That doesn't help us with our long taking file operation though and it's important to understand that this operation is not handled by the event loop, just the callback that we might have defined on write file once it's done, that code will be handled in event loop but that code will finish fast, so basically the event loop will only handle callbacks that contain fast finishing code.

Instead our file system operation and a couple of other long taking operations are sent to a **worker pool** which is also spun up and managed by Node.js automatically.



This worker pool is responsible for all the heavy lifting, this worker pool is kind of totally detached of your JavaScript code you could say and it **runs on different threads**, it can spin up multiple threads, it's closely intervened with your operating system you're running the app on, so this is really detached from your code and this worker pool is therefore doing all the heavy lifting.

Ex - If you're doing something with a file, well a worker from that pool will take care and will do its job totally detached from your code and from the request and from the event loop.

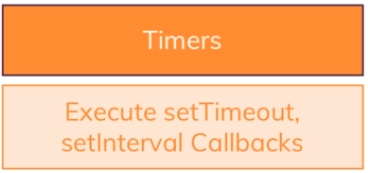


The one connection to the event loop will have, though is that once the worker is done, so for example once we read a file, it will trigger the callback for that read file operation and since the event loop is responsible for the events and the callbacks, this will in the end up in the event loop, so there Node.js will then basically execute the appropriate callback.

\*\* Now this is a lot of behind the scenes stuff which is nice to know, you don't have to write any code to make this work, this is all built into Node.js



That event loop is in the end a loop which is run or started by Node.js that keeps the Node.js process running and as I just mentioned, that handles all the callbacks and it has a certain order in which it goes through the callbacks. So basically it's a loop that just well keeps on looping unsurprisingly.



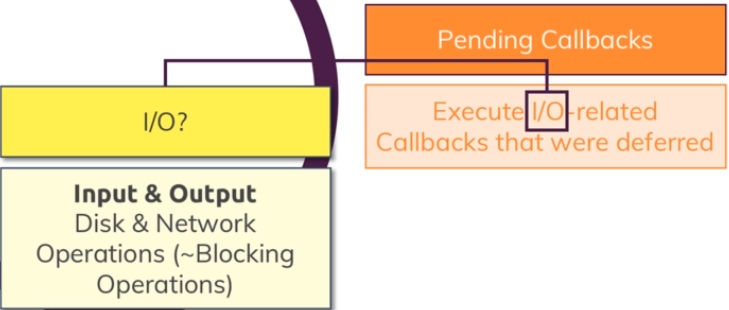
Now it's important to understand that Node.js will leave that phase at a certain point of time and that can also mean that if there are too many outstanding callbacks, it will continue its loop iteration and postpone these callbacks to the next iteration to execute them.

Then as a next step, it checks other callbacks.

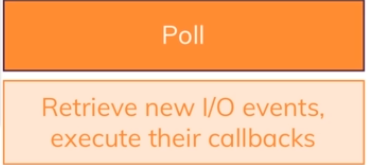
Ex - if we had write or read file, we might have a callback because that operation finished and it will then also execute these callbacks.

At the beginning of each new iteration it checks if there are any timer callbacks it should execute (setTimeout & setInterval).

We set a timer and always pass a method, a function that should be executed once that timer completes and Node.js is aware of this and at the beginning of each new loop iteration, it executes any due timer callbacks, so any callbacks that have to be executed because a timer completes.

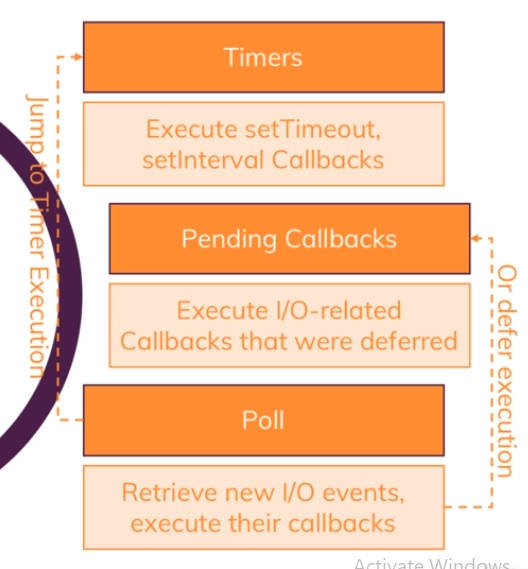


After working on these open callbacks and hopefully finishing them all, it will enter a **poll phase.**

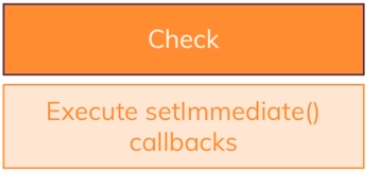


The pull phase is basically a phase where Node.js will look for new IO events, and basically do its best to execute their callbacks immediately if possible.

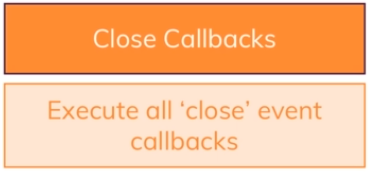
Now if that's not possible, it will defer the execution and basically register this as a pending callback.



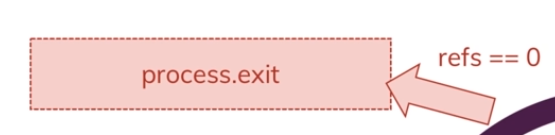
Important, it also will check if there are any timer callbacks due to be executed and if that is the case, it will jump to that timer phase and execute them right away, so it can actually jump back there and not finish the iteration, otherwise it will continue and next set immediate callbacks will be executed in a so-called **check phase**.



setImmediate() is a bit like setTimeout or setInterval, just that it will execute immediately but always after any open callbacks have been executed, so typically faster than set timeout with one millisecond of open duration, let's say but after the current cycle has finished or at least finished open callbacks that were due to be handled in that current iteration.



Now we're nearing the end of each iteration cycle and now Node.js will execute all close event callbacks, so if you registered any close events, this would be the point of time where Node.js executes their appropriate callbacks.



Well and then we might exit the whole Node.js program but only if there are no remaining event-handlers which are registered and that is what I mean with this refs equal null thing here.

Internally, Node.js keeps track of its open event listeners and it basically has a counter, references or refs which it increments by 1 for every new callback that is registered, every new event listener that is registered, so every new future work that it has to do you could say and it reduces that counter by 1 for every event listener that it doesn't need anymore, every callback it finished and since in a server environment, we create a server with create server and then listen to incoming requests with listen, this is an event which never is finished by default and therefore, we always have at least one reference and therefore **we don't exit in a normal node web server program**. We can call the exit function as we had done before.

Now I also touched on that security thing with that one JavaScript thread and regarding that, we also got no problems by default, though I will come back later to how we can manage global data and how we should manage it to ensure that we do not spoil data from request A to request B.

By default we have some separation because remember that callback method in create server gets executed for every new incoming request and therefore this function only runs for that incoming request and anything we do to the request or response object there will not be exposed to our request or response objects because each function is only scoped to itself and not accessible by the other functions, so by default we have that separation due to how JavaScript works.