How Node.js Works -

Node Architecture:

Node.js dependancies:

- V8 (C++) — Google
  - ↓ convert into machine code.
- libuv (C++)
  - ↳ open source
  - ↳ deal asynchronous -
    - ↳ file-System
    - ↳ OS
    - ↳ networking.

↓↓

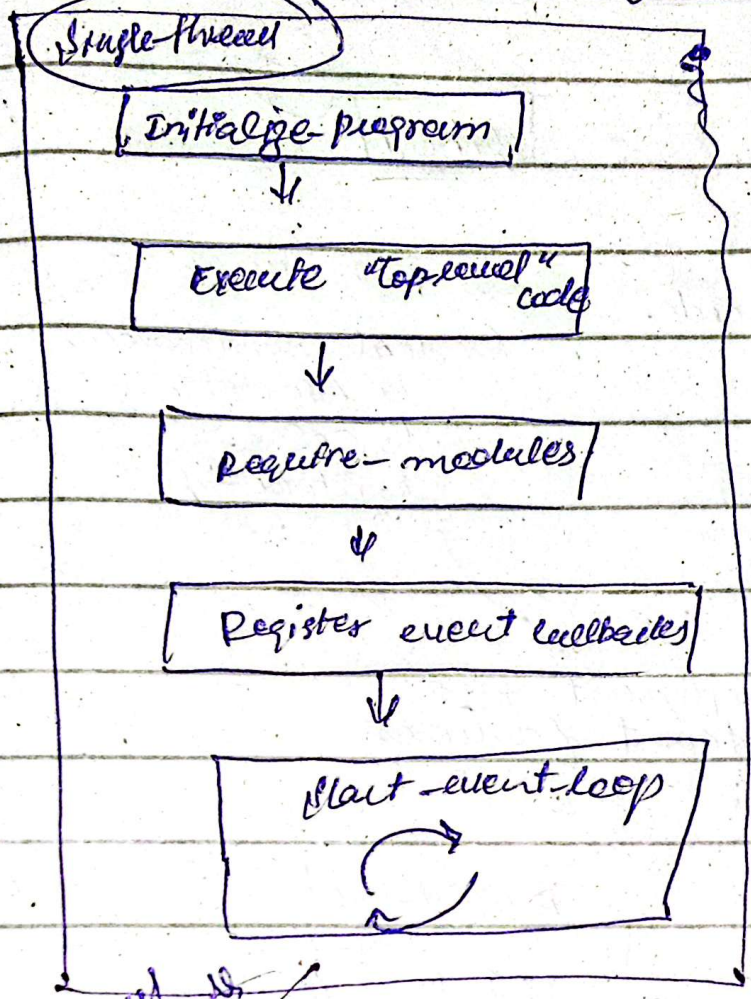
Also Implement two Important features

- Event-loop
  - ↓↓
  - * Responsible for easy tasks e.g; callbacks network-IO
- Thread-pool (C/C++)
  - T1 T2 T3 T4
  - ↓↓
  - * For heavy tasks.

- http-parser — parse http
- c-ares — DNS
- OpenSSL
- zlib

Node.js → basically cpp program.

↳ Node.js runs in single thread.



auto matically used based tasks

Same Tasks are too heavy that they can block event loop

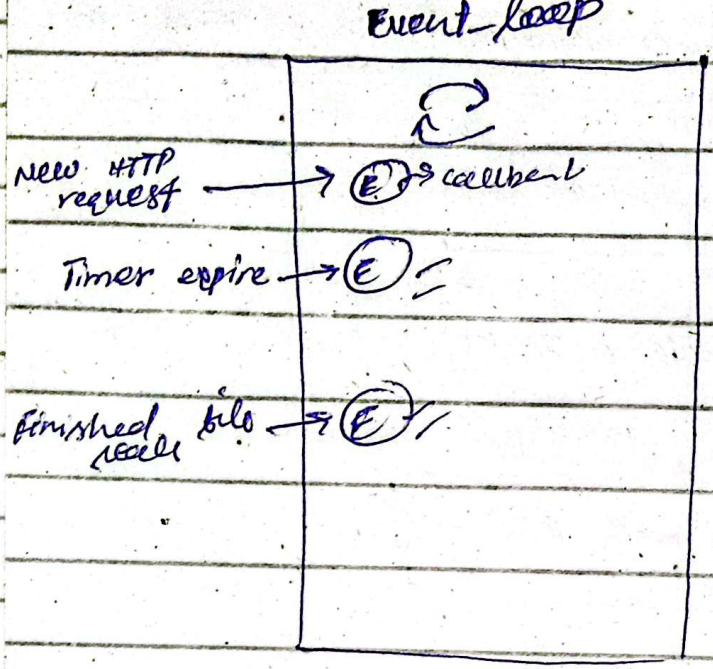
file system
+ cryptography
+ compression
+ DNS lookup

Thread-pool ← library

Provide 4 threads → 1 or more

← library

EVENT-LOOP



Event-loop

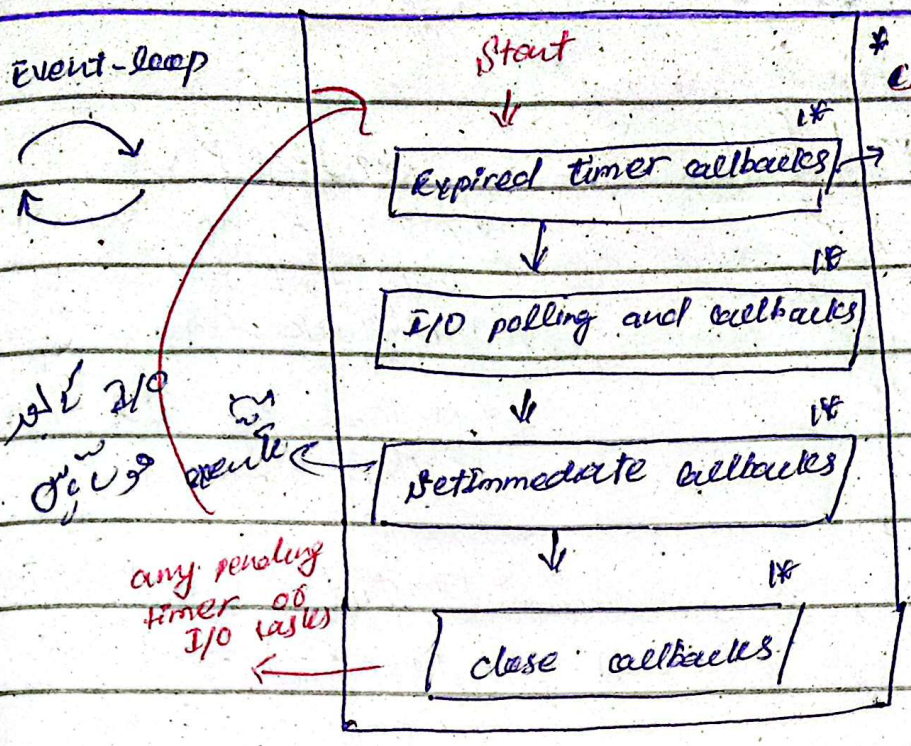
* All the application code that is inside callback function (no-blocking level)
* Node.js is build around callback function.
* Event-driven architecture
* Event are emitted.
* Event loop picks
* Callbacks are called.

* Event loop does orchestration

Event loop receive event, run their callbacks and offload more expensive to thread-pool.

* Event-loop has multiple phases. Each phase has ques

1 Tick ⇒ 1 cycle



* CALLBACK QUEUES

Expired timers callbacks are first execute

PROCESS.NEXTTICK(). QUEUE ④

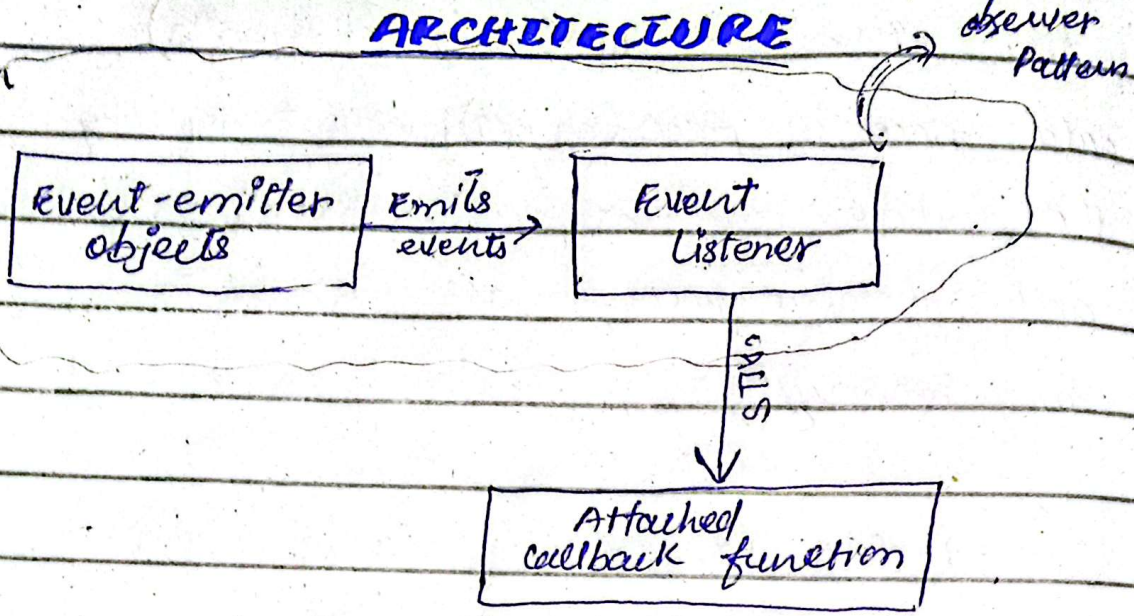
OTHER MICROTASKS QUEUE ⑤ (Resolved promises)

BEST-PRACTICES

* Don't use sync version of functions in fs, crypto, zlib in callback function.
* Don't perform complex calculations
* Be careful with JSON in large objects.
* Don't use too complex regular expressions.

observer pattern

EVENTS AND EVENT-DRIVEN ARCHITECTURE




```
const server = http.createServer();

server.on('request', (req, res) => {
    console.log("Request-recieved");
    res.end("Request recieved");
});
```

127.0.0.1:8000

Emitter

STREAMS :-

"Used to process (read & write) data piece by piece (chunks), without completing the whole read or write operation, and therefore keeping all the data in memory"

* Perfect for handling large volumes of data, for example videos.
* More efficient data processing in terms of (memory) (no need to keep all data in memory) and (time) (we don't have to wait until all the data is available)

How Streams are implemented in Node.js :-

four fundamental types of streams.

⇒ Streams are instances of the EventEmitter class!

(1) Readable Stream :-

* These are streams from which data can be read (consumed).
* HTTP requests, file system (fs) read streams.

Important Events:

data: Trigger when data is available to read.

end: Trigger when the stream has no more data.

Important Function:

Pipe(): Directly pipes the output of a readable stream into a writable stream.

read(): Reads data from the stream manually.

2 Writable Streams:

* Stream where data can be written.
* HTTP responses, file-system (fs) write streams.

Important events:

drain : Triggered when the internal buffer is empty and ready for more data.

finish : Triggered when all data has been written.

Important Functions:

write(): write data to stream

end(): Signals that no more data will be written.

③ Duplex Streams:

Streams that can both read and write data.

e.g: Web sockets (used for real time communication)

④ Transform Streams:

A special type of duplex stream that process data as it is read or written.

e.g: Gzip compression using zlib. (data is compressed before being written)

How Requiring Modules Really works

* Each JavaScript file is treated as separate module.
* Node.js uses the commonJs module System: require(), exports or module.exports
* ES module is used in browser:
  import/export;
* There have been attempts to bring ES modules to node.js (.mjs)

require('test-module');

↳ Where does it come from? ③

what happens when we Require() A module;

1) Resolving & Loading:

* Core modules
  ↳ require('http');
* Developer modules
  ↳ require('./lib/controller');
* 3rd party modules
  ↳ require("express"

* Start with core modules
* If begin with './' or '../' Try to load developer module;
* If no file found try to load index.js inside folder.
* Else goto node_modules and find module there

④

2) Wrapping :-

```
( function (exports, require, module, __filename, __dirname)
{
    // Module Code lives here
});
```

← Node.js runtime does.

ہر فائل کے اندر یہ ہوتا ہے
یہ function اس لیے nodeجس
کو wrape کرتی ہے۔

WHAT HAPPENS WHEN WE REQUIRE() A MODULE

```
require('test-module');
```

👉 **Core modules**

```
require('http');
```

👉 **Developer modules**

```
require('./lib/controller');
```

👉 **3rd-party modules (from NPM)**

```
require('express');
```

PATH RESOLVING: HOW NODE DECIDES WHICH MODULE TO LOAD

1. Start with **core modules**;
2. If begins with ‘`./`‘ or ‘`../`‘ 👉 Try to **load developer module**;
3. If no file found 👉 Try to **find folder** with `index.js` in it;
4. Else 👉 Go to **`node_modules/`** and try to find module there.

WHAT HAPPENS WHEN WE REQUIRE() A MODULE

```
require('test-module');
```


```
(function(exports, require, module, __filename, __dirname) {
    // Module code lives here...
});
```

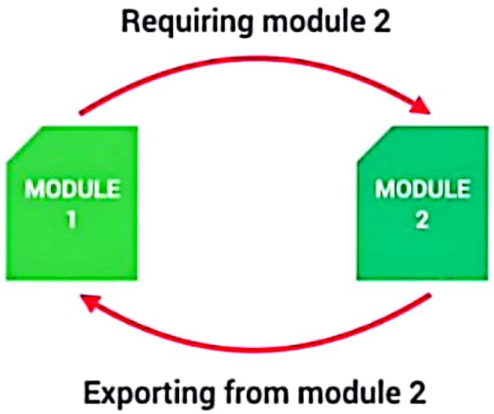
- **`require`**: function to require modules;
- **`module`**: reference to the current module;
- **`exports`**: a reference to `module.exports`, used to export object from a module;
- **`__filename`**: absolute path of the current module's file;
- **`__dirname`**: directory name of the current module.

WHAT HAPPENS WHEN WE REQUIRE() A MODULE

```
require('test-module');
```

RESOLVING & LOADING → WRAPPING → EXECUTION → RETURNING EXPORTS → CACHING

- `require` function returns **exports** of the required module;
- `module.exports` is the returned object (important!);
- Use `module.exports` to export one single variable, e.g. one class or one function (`module.exports = Calculator`);
- Use `exports` to export multiple named variables (`exports.add = (a, b) => a + b`);



WHAT HAPPENS WHEN WE REQUIRE() A MODULE

```
require('test-module');
```