**Node JS**

* Graphical user interface, application

  Description automatically generated
  + Most Important dependencies of Node JS is the chrome **V8 engine** and **libuv**.
  + V8 engine converts the **JS code to machine code.**
  + Libuv has focus on **Asynchronous IO**. This gives nodejs access to hardware and resources. The **event loop** and the **thread pool** are contained in libuv.
  + Libuv is completely written in C++. V8 is written in both V8 and JS.
  + Beside of V8 and libuv nodejs also consists of **http-parser** (for parsing http ), **c-ares** (for DNS stuff), **OpenSSL** (for cryptography) and **zlib** (for compression).
* **NodeJS Process**
  + A picture containing timeline

    Description automatically generated
  + It is a **single thread operation**.
  + First all the computer intensive tasks are completed and then the program enters in the Event loop where now **all the requests will be received,** and **all the callbacks will be executed.**
  + Node Js also provide **min 4 and max 128 additional threads** in which CPU intensive work (like File system APIs, cryptography, compression, DNS lookups etc.) can be done and are **independent of main thread.**
* **EVENT LOOP**

**A picture containing graphical user interface

Description automatically generated**

* + Event loop receives events, call the callbacks, and offloads the CPU extensive events to thread pools.
  + All the callbacks are executed in a particular order.

**Diagram

Description automatically generated**

* + Event loop executes in many phases and each phase has its own callback queue and they are –
    - Expired Timer callbacks (eg- setTimeout())
    - I/O polling and callbacks (e.g.- readFile())
    - setImmediate callbacks
    - close callbacks (e.g. – closing of WebSocket)
  + The callback queues in each phase are executed when the event loop reaches that phase. But if the callbacks are not ready ( suppose time in the setTimeout() is not finished yet) then it will be executed in the next cycle of event loop.
  + There are some special callback queues which are executed after each phase and they are
    - PROCESS.NEXTTICK() Queue
    - Other MICROTSDK QUEUE (for resolving promises
  + The cycle -
  + **Diagram

    Description automatically generated**
* **How to not block event loop**
  + **Text

    Description automatically generated with medium confidence**
* **The EVENT-DRIVEN Architecture**
  + **Graphical user interface

    Description automatically generated with medium confidence**
  + There are three components of this architecture
    - * **Event emitter -** these emits named events as soon as something important happens like file read complete or timer complete.
      * **Event listener –** this will read the emitted events.
      * **Attached callback function –** these are the callback functions which the event listener will call.
  + **E.g.** **Graphical user interface

    Description automatically generated**
* **STREAMS**
  + Streams are used to process (read and write) data piece by piece (chunks), without completing the whole read or write operation, and therefore without keeping all the data in memory. E.g. Netflix and YouTube video streaming.
  + Types of streams –

Graphical user interface, application

Description automatically generated

* **Importing a module**
  + These are the steps involved when we write **require(‘module’)** .

**A screenshot of a computer

Description automatically generated with medium confidence**

* + **Step 1:**

**Graphical user interface, application

Description automatically generated**

* + **Step 2**
    - Everything we right inside a Nodejs file is wrapped inside a wrapper function having the arguments shown in the image.
    - If we console.log(arguments) we will able to see every one of them.

**Graphical user interface, application

Description automatically generated**

* + **Step 3**
    - The requires module is executed.
  + **Step 4**

**Graphical user interface, Teams

Description automatically generated with medium confidence**

* + **Step 5** 
    - If a module is used multiple time, it is cashed so that it can be retrieved fast.
* **Environment variables**
  + There are two ways to add env variables in our code.
    - By adding the variables in the run script each time.
      * E.g., NODE\_ENV=development X=23 nodemon server.js
    - By adding a **config.env** file and add them their directly and then install a npm module **dotenv.**
      * Then require the dotenv in the root file and use

**dotenv.config({ path:** 'config file path' **})**

* **Code linting**
  + We use code linting to show errors when writing the code only, which can save us from potential errors.
  + To do that we need to first install two extensions in VS code
    - **Prettier - Code formatter**
    - **ESLint**
  + Then we must install these packages in npm dev dependencies.( npm i eslint eslint-config-airbnb eslint-config-prettier eslint-plugin-import eslint-plugin-jsx-a11y eslint-plugin-node eslint-plugin-prettier eslint-plugin-react prettier)
    - **eslint**
    - **eslint-config-airbnb**
    - **eslint-config-prettier**
    - **eslint-plugin-import**
    - **eslint-plugin-jsx-a11y**
    - **eslint-plugin-node**
    - **eslint-plugin-prettier**
    - **eslint-plugin-react**
    - **prettier**
  + Once installation done add the prettier and eslint config files and restart the workspace.
  + This setup will help in detecting code errors which we will add in the eslint config . E.g., variable declared but not used, console.log() etc.