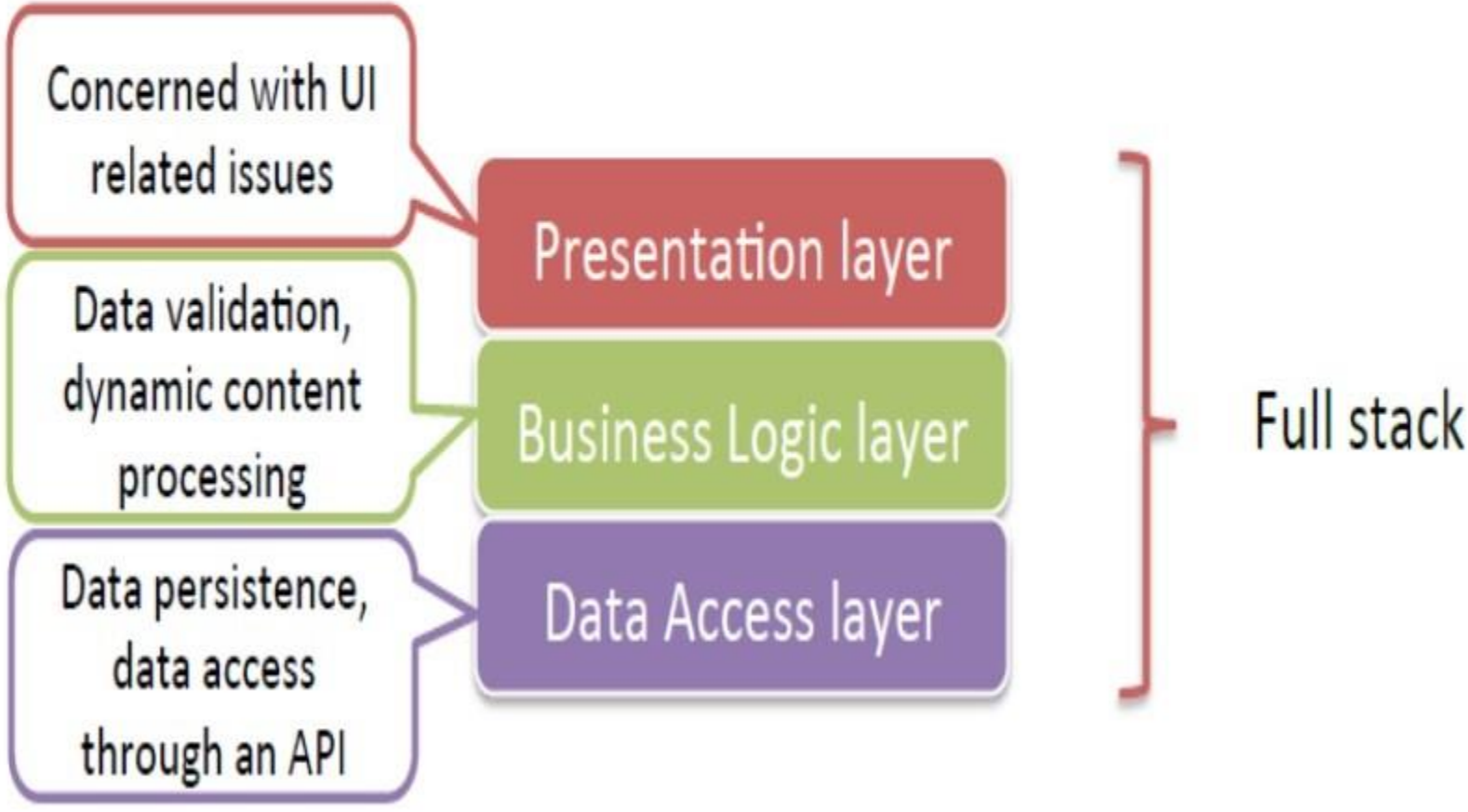
NODE.JS

Few Terms and Terminologies

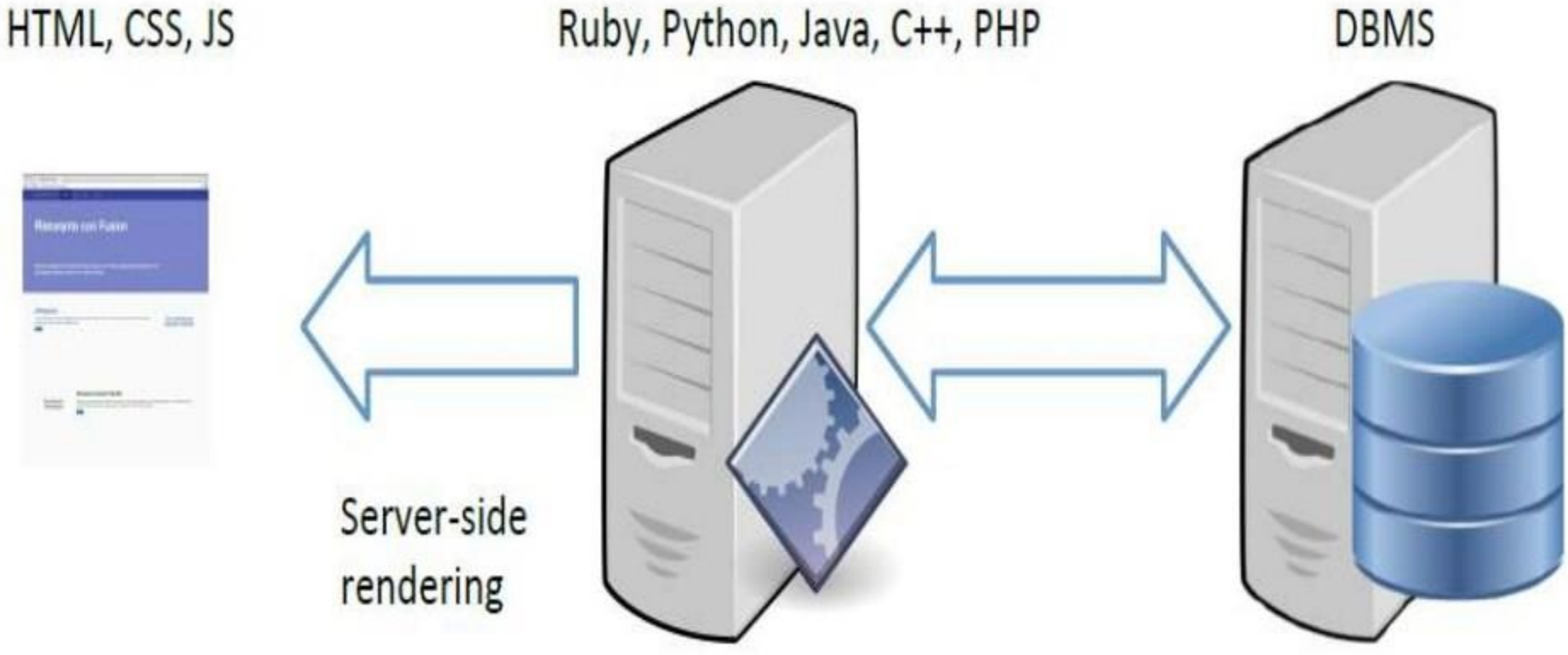
Front end and Back end

- Front end / Client-side
  - HTML, CSS and Javascript
- Back end / Server-side
  - Various technologies and approaches
  - PHP, Java, ASP.NET, Ruby, Python

Three Tier Architecture



Traditional Web Development

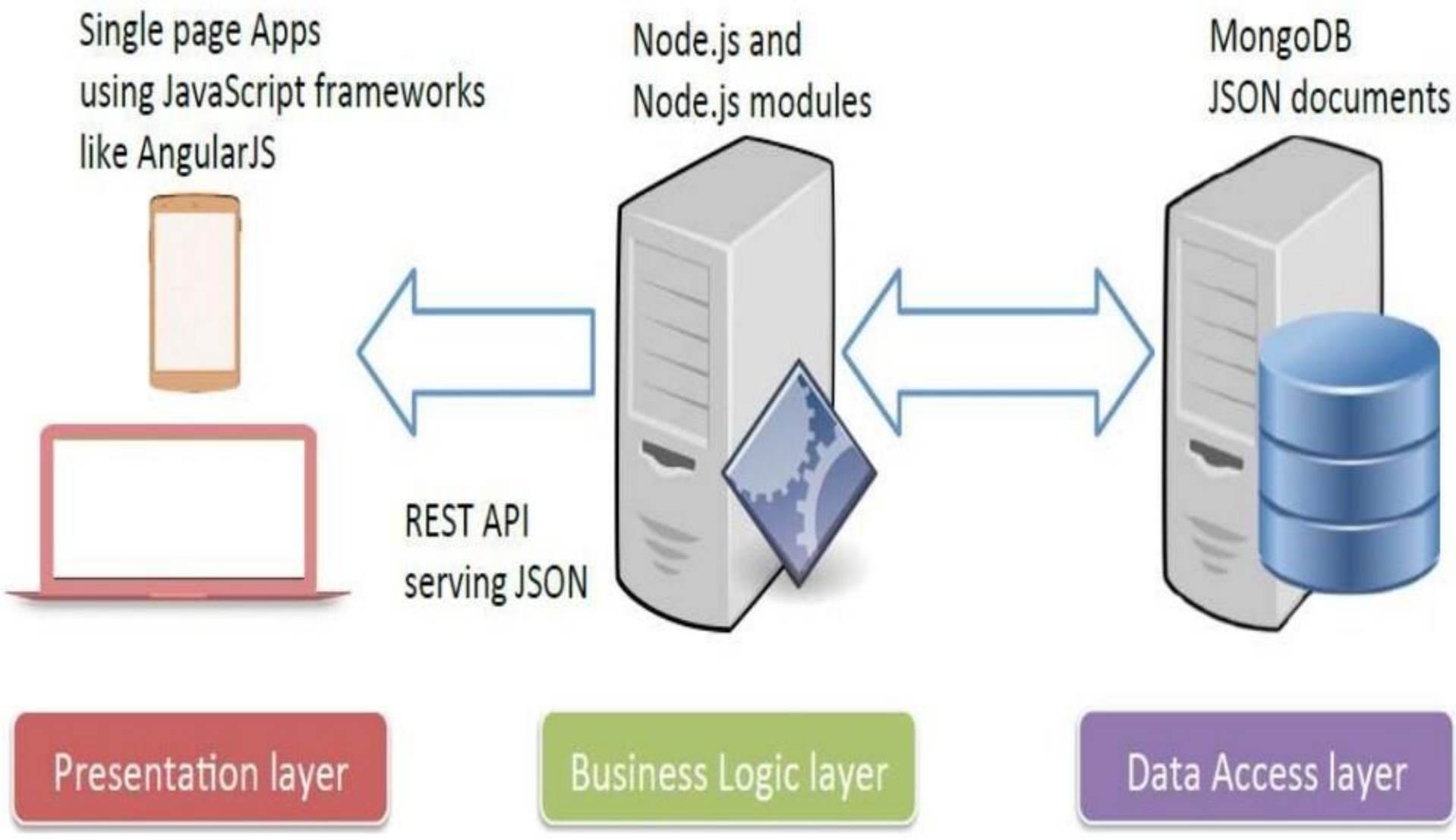


Presentation layer

Business Logic layer

Data Access layer

Full Stack JavaScript Development

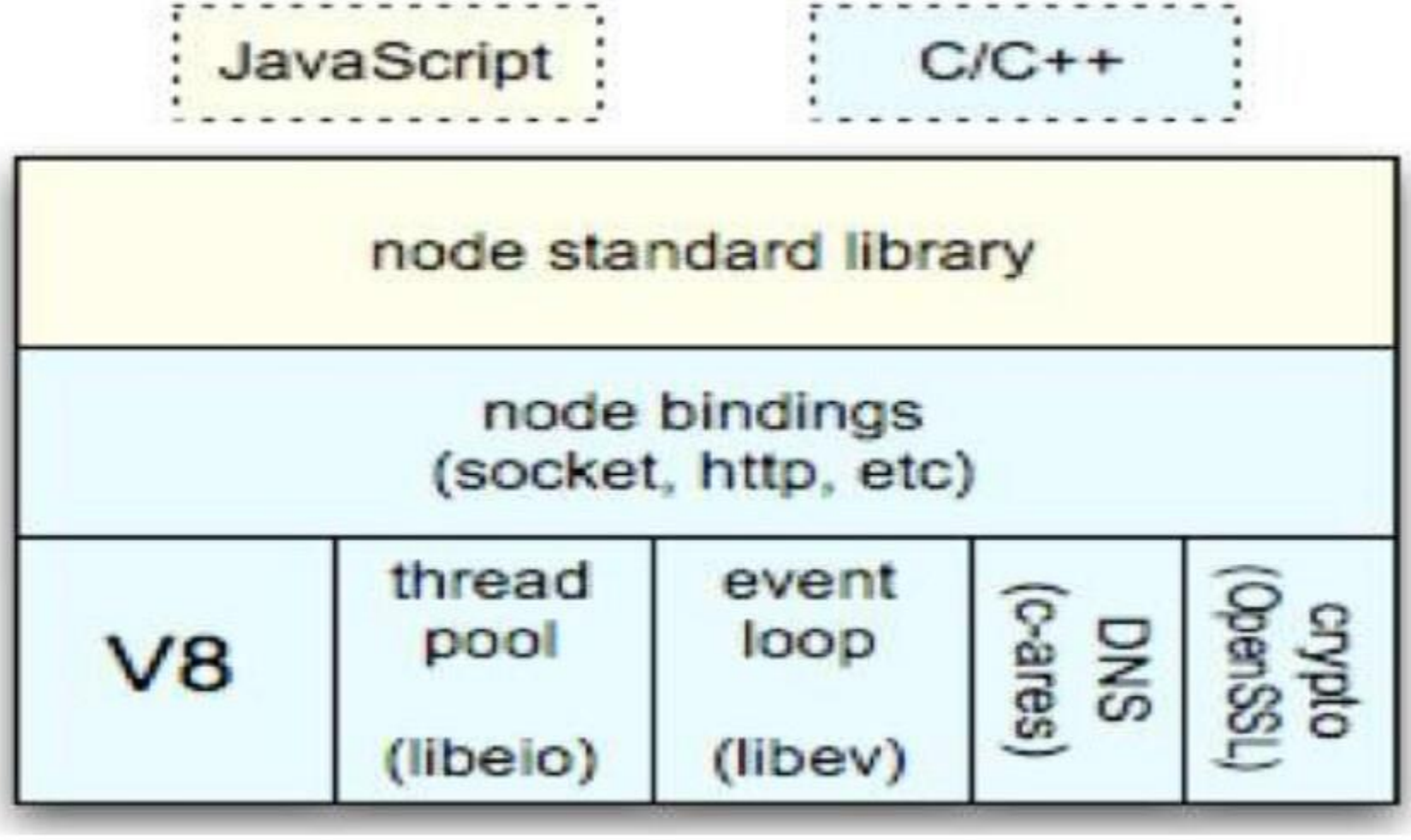


Node.js

Introduction

- Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.
- Node.js was developed by Ryan Dahl in 2009 and its latest version is v7.0.0.

Node.js architecture



Features of Node.js

- Powerful js framework which works on cross platform
- Asynchronous and Event Driven

All APIs of Node.js library are asynchronous, that is, non-blocking.

- Very Fast

Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.

- Single Threaded but Highly Scalable

Node.js uses a single threaded model with event looping

- No Buffering

Node.js applications never buffer any data. These applications simply output the data in chunks.

- Node.js is released under the MIT license

When to use

- When making system which put emphasis on concurrency and speed.
- Sockets only servers like chat apps, irc apps, etc.
- Social networks which put emphasis on realtime resources like geolocation, video stream, audio stream, etc.
- Handling small chunks of data really fast like an analytics webapp.

Where Not to Use Node.js?

It is not advisable to use Node.js for CPU intensive applications.

http://www.algoworks.com/blog/developing-enterprise-applications-using-node-js/

Following are the sample applications for which node.js is much suitable:

- I/O based Applications
- Data Streaming Applications
- Data Intensive Real-time Applications (DIRT)
- JSON APIs based Applications
- Single Page Applications

V8 --->Javascript Engine

- It is a open source javascript engine written in C++.
- It is used in client side in Chrome browser and server side in NodeJs.

- V8 can run standalone and can be embedded into any C++ application.
- Its was designed to increase performance of Javascript execution inside web browsers.

- V8 translates JavaScript code into more efficient machine code instead of using an interpreter and does not produce byte code or any intermediate code.

- **To Check Version of V8 installed:**

- **Type in terminal :**

- **node -p process.versions.v8**

NodeJS Installation

- Installing On Linux:

  sudo apt-get update
  sudo apt-get install nodejs
  sudo apt-get install npm

- **Just close the terminal and reopen it and type:**

node –v

- Install on Windows:

Download the MSI for Node and install it and follow the instructions.
Open The CMD and type node -v, it should say version of nodejs.

NodeJS REPL Mode

REPL stands for Read Eval Print Loop and it represents a computer environment like a Windows console or Unix/Linux shell where a command is entered and the system responds with an output in an interactive mode.

- There are a few REPL-specific commands available to you, which can be displayed by typing: **.help**

```
> 3+1
4
> x=3
3
> var x=3
undefined
> console.log('Hello World')
Hello World
undefined
> var http= require('http')
undefined
```

REPL Commands

- **ctrl + c** − terminate the current command.
- **ctrl + c twice** − terminate the Node REPL.
- **ctrl + d** − terminate the Node REPL.
- **Up/Down Keys** − see command history and modify previous commands.
- **tab Keys** − list of current commands.
- **.help** − list of all commands.
- **.save *filename*** − save the current Node REPL session to a file.

Node JS REPL mode....

- We can change default prompt(>) to custom :
  node -e "require('repl').start({prompt : 'username>>'})"

- We can ignore Undefined returned REPL in node in case of statement which return nothing like, var x=9; As javascript returns something if nothing is returned it returns Undefined as value. To ignore it we can type:
  node -e "require('repl').start({prompt : 'username>>>',ignoreUndefined : true})"
- Run example code **repl_output.js** to see case change of output in console.

Add New Commands to REPL

- Type .help to see available default commands in REPL mode.
- Demo code to add new command:

replcommand1.js

replcommand2.js

Type to run**:**

node replcommand1.js

.help (to see command added in list)

Creating a simple server with node Environment

A Node.js application consists of the following three important components −
1) Import required modules − We use the **require** directive to load Node.js modules.

2) Create server − A server which will listen to client's requests similar to Apache HTTP Server.

3) Read request and return response − The server created in an earlier step will read the HTTP request made by the client which can be a browser or a console and return the response.

Demo Example: helloworld.js

Node Package Manager (NPM)

Node Package Manager (NPM) provides two main functionalities −

- Online repositories for node.js packages/modules which are searchable on search.nodejs.org
- Command line utility to install Node.js packages, do version management and dependency management of Node.js packages.

NodeJS and NPM

- It is automatically installed with Node.js, and we use NPM to install new modules.
- It is a Node module that is installed globally with the initial installation of Node. By default, it searches and loads packages from the npm registry.
- To install a module, open your terminal/command line, navigate to the desired folder, and execute the following command:

npm install module_name

Simply executing **npm install** will install all the modules listed in the **package.json** file in local local **node_modules** folder in the current directory.

- **npm install <name>** tries to install the most recent version of the **module <name>** into the local **node_modules** folder.

NodejS and NPM .....

//To check npm version
npm –version

//To check all modules installed in current directory
npm ls

//To check modules installed globally
npm ls –g

//To uninstall a module
npm uninstall <module_name>

//To uninstall a update
npm update <module_name>

//To search a module
npm search <module_name>

//To Interactively create a package.json
npm init

//To automate publishing of **npm** modules
npm publish

NPM and Package.Json

```
{
  "name": "hr",
  "version": "1.0.0",
  "scripts": {
    "start": "echo \"Run the correct js file to start your
➥application\" && exit 0",
    "populate": "node ./bin/populate_db"
  },
  "dependencies": {
    "async": "0.9.0",
    "debug": "0.7.4",
    "express": "4.2.0",
    "mongoose": "3.8.11"
  }
}
```

NPM and Package.json ….

- **name**: is the name of the module. This is required, but only needs to be unique if you plan on publishing the module to the npm registry.
- **version:** is the version of the package expressed as **major.minor.patch**. This is known as semantic versioning, and it is recommended to start with version 1.0.0.
- **scripts:** This key serves a unique purpose, providing additional npm commands when running from this directory.
- **dependencies**: This is generally where the majority of information in any package.json file is stored. dependencies lists all the modules that the current module or application needs to function.
- Run **npm init** in the terminal and it will guide you through a series of prompts. This will generate a simple package.json file that's in line with the community standard and the latest specification from npm.
- When there is a **package.json file** , running npm install module_name will install that module and update dependencies in package.json file to include an entry for those module.

NPM require() function

- The require function is specific to Node and is unavailable to web browsers. **require('http')** will cause Node to try to locate a module named **http**. The lookup procedure roughly works in this way:

1. Check to see if the module named is a core module.
2. Check in the current directory's node_modules folder.
3. Move up one directory and look inside the node_modules folder if present.
4. Repeat until the root directory is reached.
5. Check in the global directory.
6. Throw an error because the module could not be found.

Modules

- Node.js uses a module architecture to simplify the creation of complex applications. Modules are akin to libraries in C.
- Then there are modules available for social authentication, validation, and even server frameworks such as ExpressJS and Hapi.
- Each module contains a set of functions related to the "subject" of the module.
- For example, the http module contains functions specific to HTTP.

Modules .....

- Including a module is easy, simply call the require() function, like this:

  var http = require('http');

- The require() function returns the reference to the specified module.
- This causes Node to search for a node_modules folder in our application's directory, and search for the http module in that folder. If Node does not find the node_modules folder (or the http module within it), it then looks through the global module cache.

Writing Module locally

1) Create folder mmm Write a file save it as
mmm.js
2) Go to mmm folder and Run npm init ,
it will prompt you for values for the package.json fields. The two required fields are name and version.
3) Go to mmm folder and Run npm pack
This pack command creates mmm-1.0.0.tgz file (this is your node packaged module locally)
4) To install it , just go to any folder and run
npm install (path to mmm-1.0.0.tgz file)
ex: >npm install ..\mod\mmm-1.0.0.tgz
5) To test the newly installed module create **test.js and run it.**

Writing Module Globally

1) Create a folder
2) Go to folder and run : npm init
 name is mandatory should keep unique and press enter to set default values.
3) Create a new git repository and upload files there, by default README.md file will create in GIT.
4) Now edit package.json file in git repository as well as in folder you have made and add the

```
"repository": {
    "type": "git",
    "url": "https://github.com/kpriyacdac/kpriyamodule"
  },
```

5) Run in same folder you made : npm adduser

*** **npm install looks for package.json file and install alll dependencies . If not found package.json file it gives a warning but makes a empty node_modules folder**

npm install <module_name> willl install this module and after that will lokk for package.json file to install or update other dependencies if any mentioned.

Event Loop

- Node.js is a single-threaded application, but it can support concurrency via the concept of **event** and **callbacks**. Every API of Node.js is asynchronous and being single-threaded, they use **async function calls** to maintain concurrency. Node uses observer pattern. Node thread keeps an event loop and whenever a task gets completed, it fires the corresponding event which signals the event-listener function to execute.
- JavaScript engine maintains several queues of unhandled tasks. These queues include things such as events, timers, intervals. Each execution of the event loop, known as a cycle, causes one or more tasks to be de-queued and executed. Demo example file: **eventloop.js**

- Nodejs almost exclusively uses asynchronous non-blocking I/O. Under this paradigm, an application will initiate some long-running external operation such as I/O, however, instead of waiting for a response, the program will continue executing additional code. Once the asynchronous operation is finished, the results are passed back to the Node application for processing.
  Few ways it can be done:
  The two most popular ways in Node are:
  1. callback functions (more priority)
  2. event emitters

What is Callback?

- Callback is an asynchronous equivalent for a function. A callback function is called at the completion of a given task. Node makes heavy use of callbacks. All the APIs of Node are written in such a way that they support callbacks.

- For example, a function to read a file may start reading file and return the control to the execution environment immediately so that the next instruction can be executed. Once file I/O is complete, it will call the callback function while passing the callback function, the content of the file as a parameter. So there is no blocking or wait for File I/O. This makes Node.js highly scalable, as it can process a high number of requests without waiting for any function to return results..

Conventions followed:

1. When passing a callback function as an argument, it should be the last argument.

2. error is the first argument to the callback function.

Demo example **syncnode.js** to see synchronous nature.

Demo example **asyncnode.js** to see asynchronous nature and callback.

Event Emitters

- The second way to implement asynchronous code is via events. Under this model, objects called event emitters create or publish events. For example, in the browser, an event could be a mouse click or key press.

Demo example of event :

event .js

Node.js - Buffers

- Node provides Buffer class which provides instances to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap.
- Buffer class is a global class that can be accessed in an application without importing the buffer module.

Example:

```
buf = new Buffer(256);
len = buf.write("Simply Easy Learning");

console.log("Octets written : "+  len);
```

Working with the File System

The **fs module** allows you to access the file system.

- __filename and __dirname are strings, and, as their names imply, they specify the file being executed and the directory containing the file. They are local variables that are defined in every file.

  Demo code**:** **file_path.js**

Reading Files:

- fs module's readFile() and readFileSync() methods. Both of these methods take a filename to read as their first argument. An optional second argument can be used to specify additional options such as the character encoding . I f the encoding is not specified, the contents of the file are returned in a Buffer (a Node data type used to store raw binary data). Example code: **readfile.js**
- There are two ways to view the data as a string**:**
  1. First is to call **toString()** on the data variable. This will return the Buffer contents as a UTF-8 encoded string.
  2. The another way to specify UTF-8 encoding using the optional second argument.

     Example file: **file.js.**

Working with the File System.....

- Compare codes **file.js and file1.js** to see difference between readFile and readFileSync methods
- To get a file info run the program: **info.js**

Writing Files:

- Files can easily be written using the **writeFile()** and **writeFileSync()** methods. These methods are the counterparts of **readFile()** and **readFileSync().**
- These methods take a filename as their first argument, and the data to write (as a string or Buffer) as their second argument. The third argument is optional, and is used to pass additional information such as the encoding type.
- Unlike the **readFile()** variations, these methods default to using UTF-8 encoding. **writeFile()** takes a callback function as its last argument.
- Example code run: **writefile.js**

Debugging

- Debugging, as the name suggests, is the process of tracking down bugs and fixing them. The process of debugging can be as simple as adding **console.log()** calls to your code to verify that certain variables hold expected values.

- All major JavaScript environments (browsers, Nodejs, and so on) come with a built-in debugger. However, the debugger is not typically enabled by default.

- JavaScript's **debugger** statement is used to invoke a debugger on an application, if one is attached. If an application has no debugger associated with it, the debugger statement has no effect.

- Open Chrome's developer tools by right-clicking on the page and clicking Inspect Element. Next, refresh the page. Doing this with the developer tools enabled allows the **debugger** to be attached.

Debugging With Chrome

- Demo Code : **debug.html**

- Open Chrome's developer tools by right-clicking on the page and clicking Inspect Element. Next, refresh the page. Doing this with the developer tools enabled allows the debugger to be attached. The execution pauses at debugger statement.
- This predefined pause in execution is known as a **breakpoint**.

- On the right-hand side of the image, notice the two panels, **Scope** Variables and **Global**. These panels can be expanded to view the variables and values in the local and global scope respectively.
- Our code is currently executing in the global scope, so nothing is listed in the Scope Variables panel. Expand the Global panel and scroll down until you find i and j.
- In right panel there are controls to resume execution step in , deactivate breakpoint. We can edit value of vriables. The value edited persist even out of debugger.
-
-
-

Node Debugging

- Node ships with a built-in debugger. To debug a file just type as below: **node debug filename**

The following table specify commands for debugging:

Command	Description
`cont` or `c`	Resumes execution
`next` or `n`	Steps to the next instruction
`step` or `s`	Steps into a function call
`kill`	Kills the executing script
`restart`	Restarts the script
`pause`	Pauses running code
`scripts`	Lists all loaded scripts
`list(n)`	Displays source code, showing `n` lines before and `n` lines after the current line