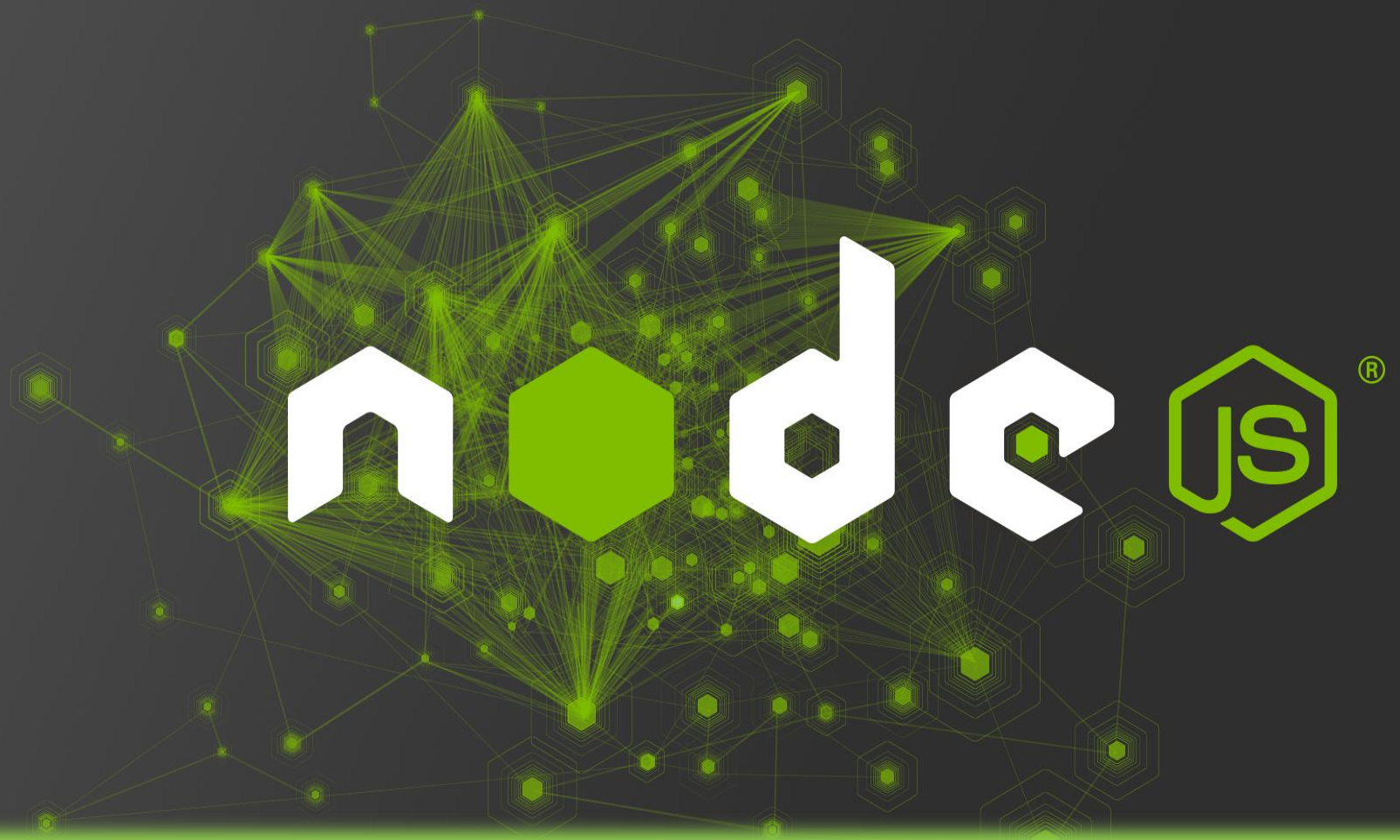


Learning server side JavaScript

Eman Fathi

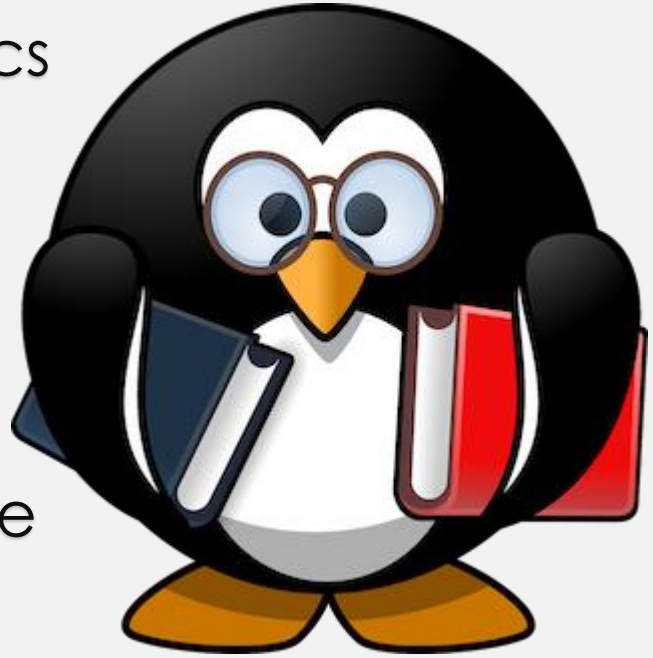
Conents

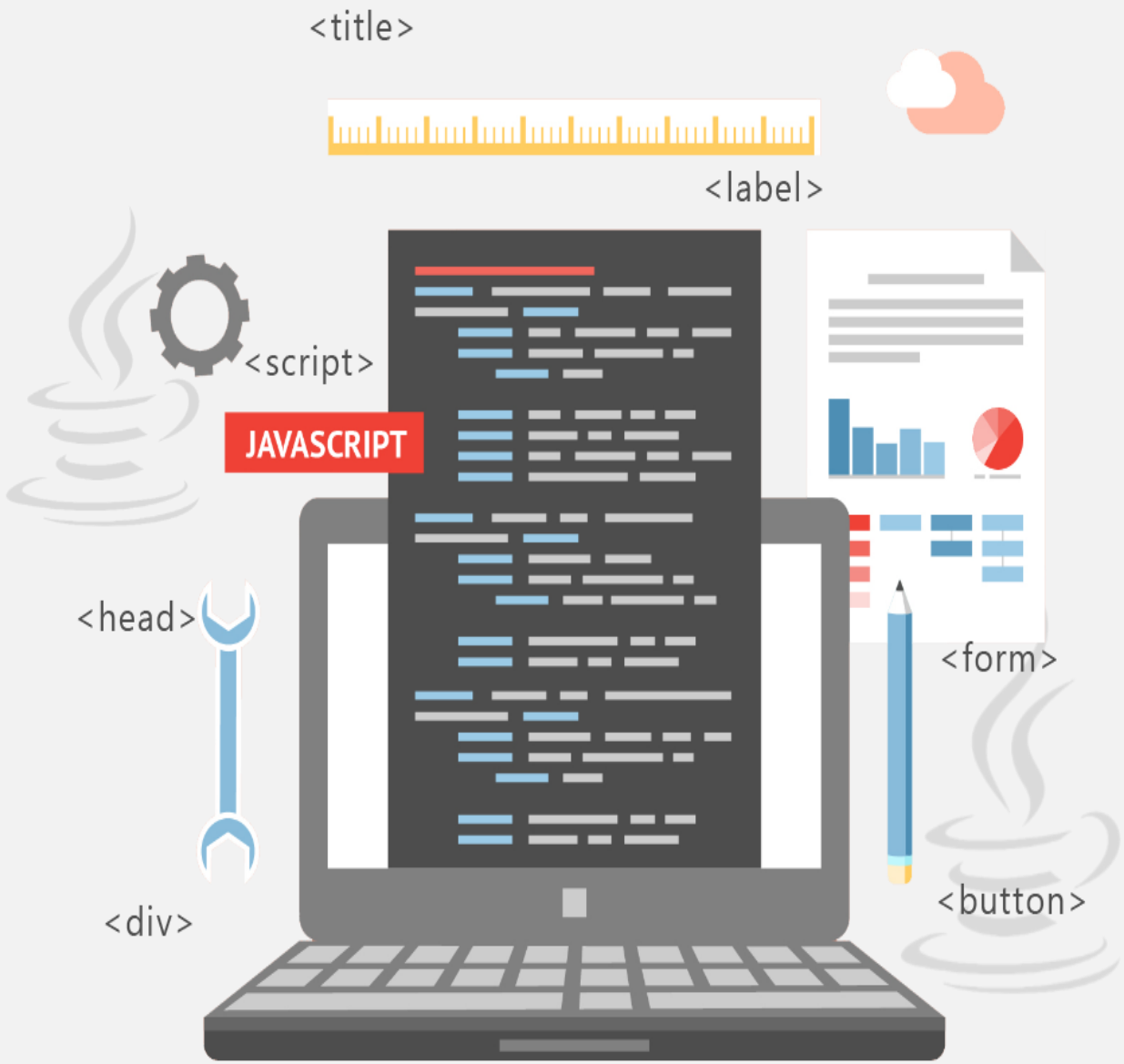
Day1 : Installation ,Nodejs structure and Module Basics

Day 2 : Express web framework basics part1

Day 3 : Express web framework basics part2

Day 4 : connect to mongo Database with mongoose





Server Side JavaScript

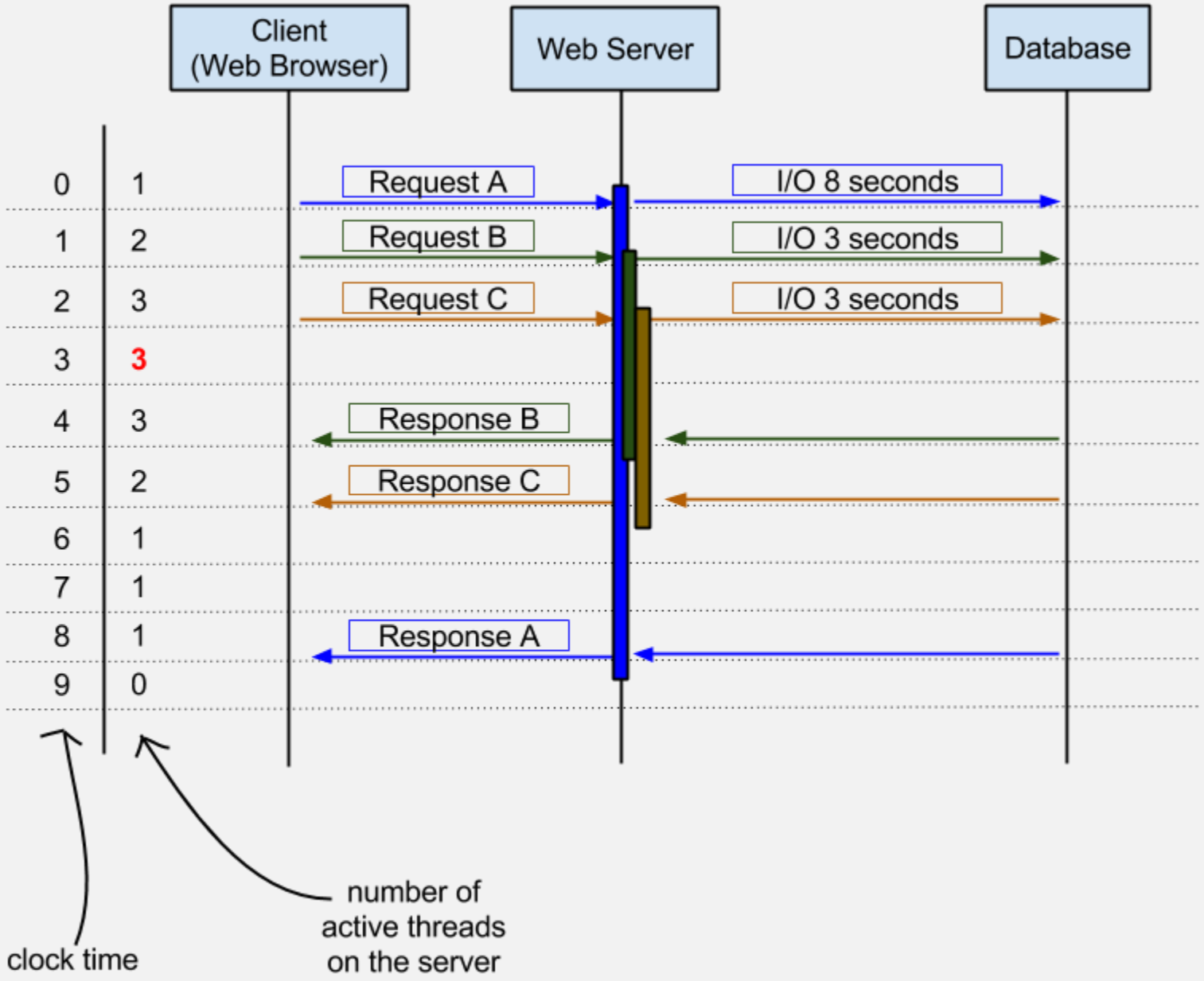
"What's a browser language doing on the server?".

JavaScript every where

JavaScript is a programming language, just like any other language, and the better question to ask is

"Why should JavaScript remain trapped inside browsers?"


Ryan Dahl
Nod.js creator 2009
Software Engineer @ Joyent



Multi-Threading Server

Nodejs Creation

- ✓Sever side JavaScript
- ✓Built on Google's V8 JavaScript Engine
- ✓Single-threading, Event loop, non-blocking IO System
- ✓CommonJs Module System
- ✓Written in C++ & javascript

NodeJs Creation

2009: Node.js was originally written by Ryan Dahl
supported only Linux and Mac OS

2011: Node Package Manager (NPM) created
and first windows version of nodejs released

2013: MEAN stack by *Valeri Karpov*

2014 : io.js project forked

2015: Node.js 4.0 released including ES6 features

V8 Google chrome JavaScript Engine

V8 is Google's open source high-performance JavaScript engine, written in C++.

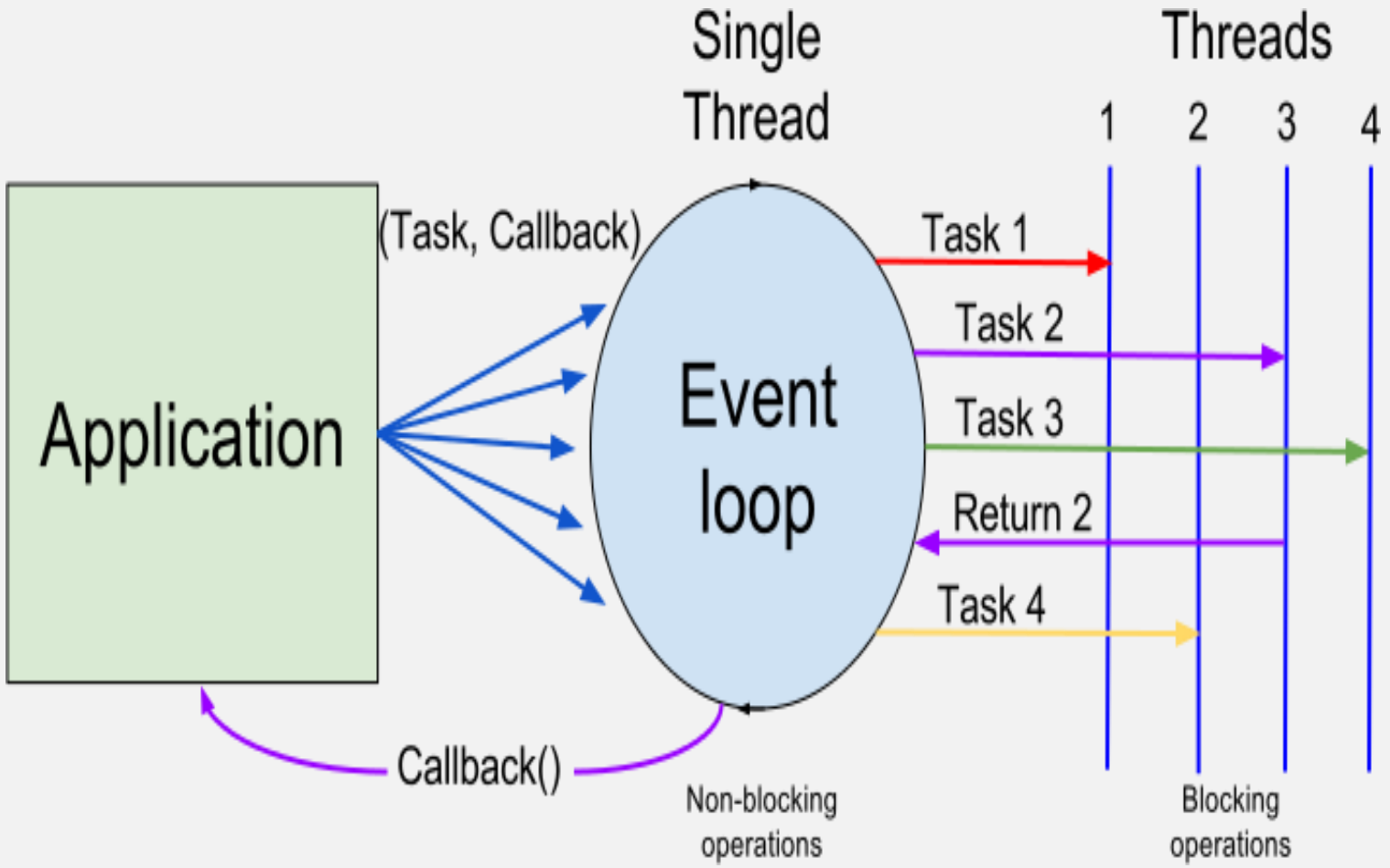
It compiles the JavaScript code into machine code at execution by implementing a JIT compiler .

Usage:

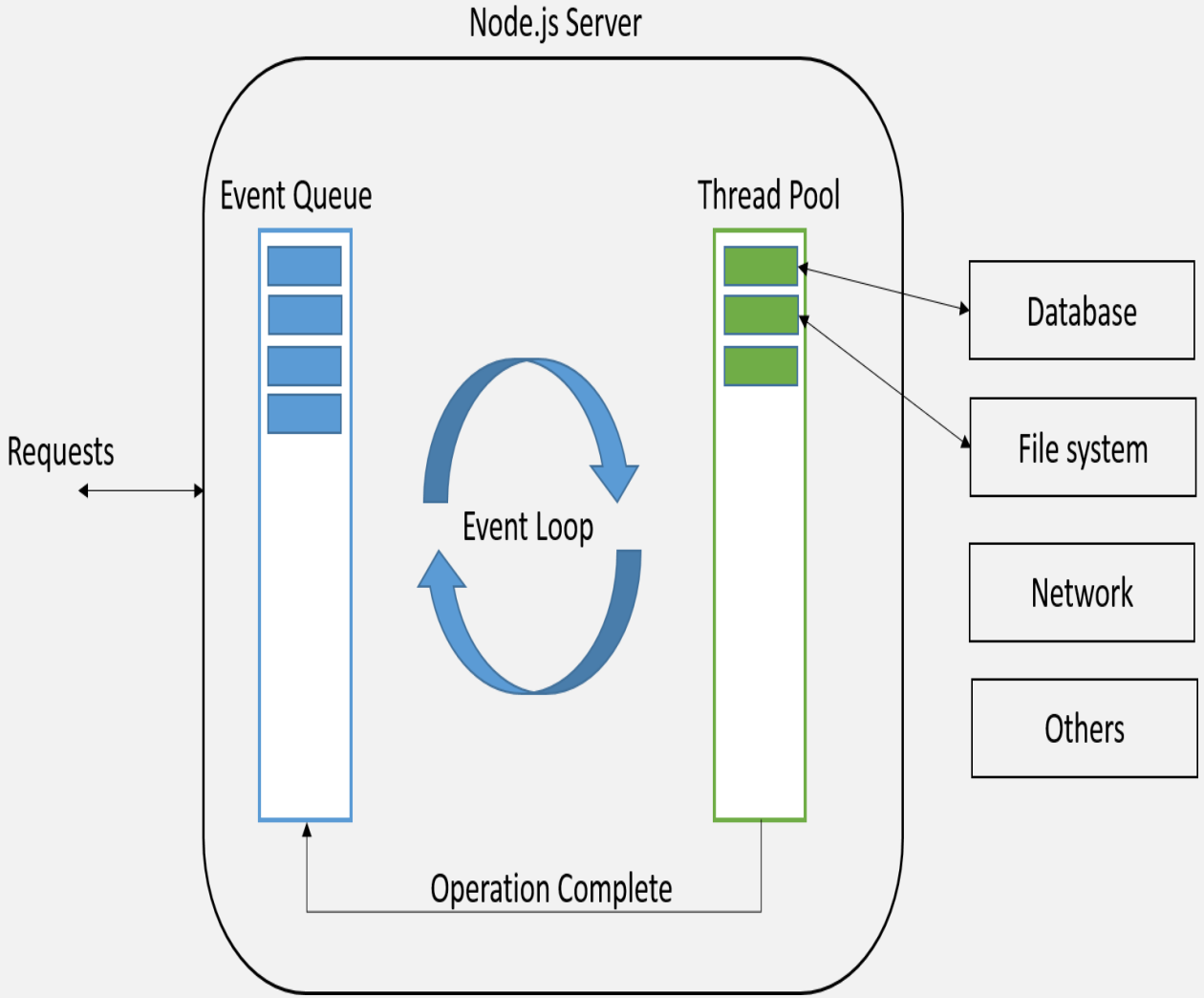
1- chrome browser

2- V8 is the core of node.js

3-Underlying component for ATOM and Visual Studio code text editors



Node.js Single-Thread Server



1- Node pass async. Task to the event loop, along with a callback

2-The event loop efficiently manages the thread pool and executes tasks

3-featch call back from event queue when task is finished to execute

Sever side JavaScript is Single-Threaded Non-blocking Asynchronous Language

Sever side JavaScript is:
Single-Threaded
Non-blocking
Asynchronous Language

- ✓ JavaScript is synchronous (one process at a time)
- ✓ JavaScript can run as asynchronous :
  - Callback functions
  - Promises
  - Async/await

Callback functions in JavaScript

```
function callbackFunction()
{
        console.log("Timeout");
}
setTimeout(callbackFunction(),1000);
```

```
$("button").on("click",function(){
        //do something here
})
```

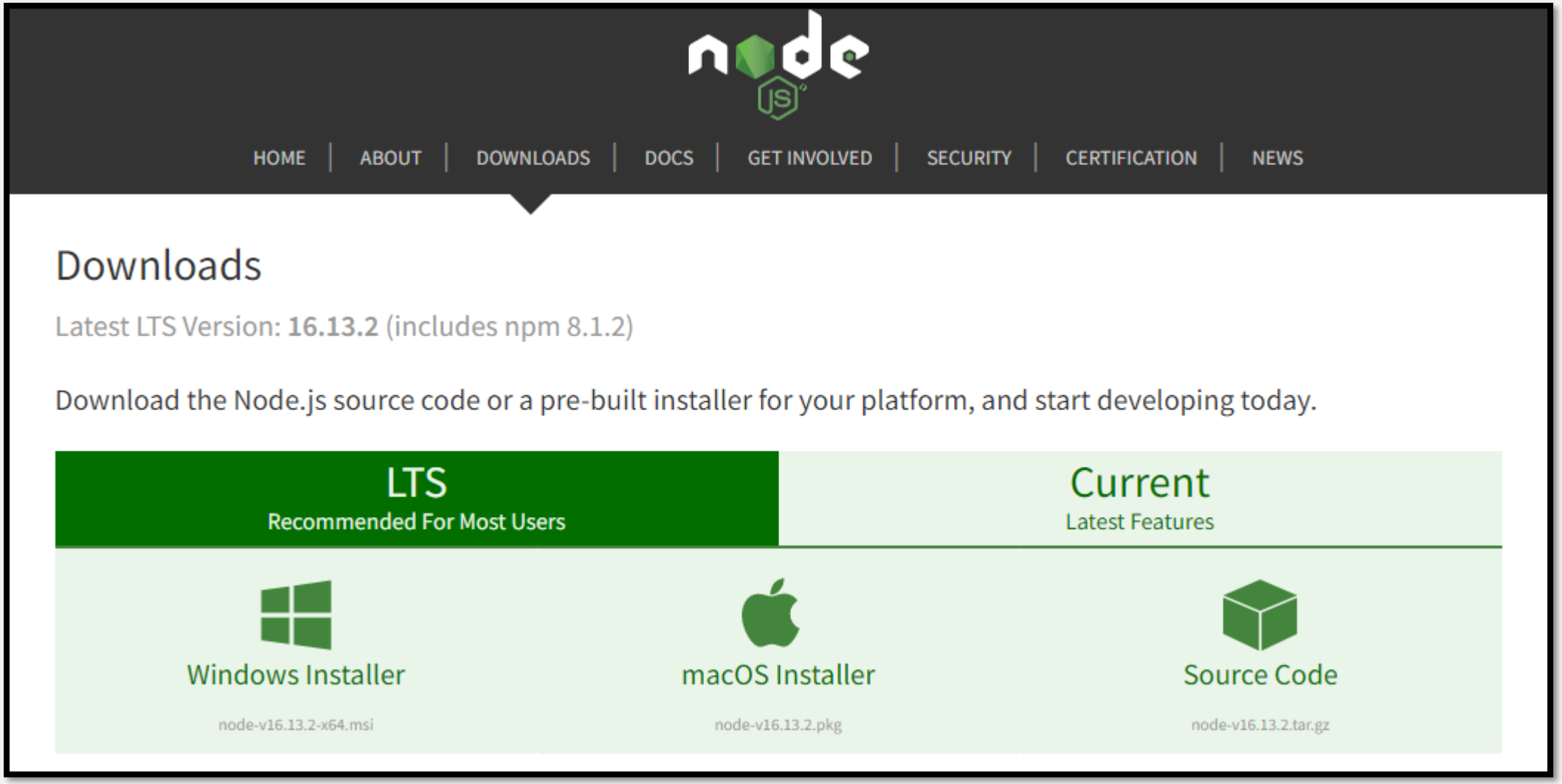
```
[3,2,5,1].filter(function(item){
        //write filteration code here
});
```

What Does JavaScript need to run as Server?

- ✓Organize code into reusable code.
- ✓Dealing with the requests over the internet .
- ✓Ability to accept request and send responses back to the client.
- ✓Ability to deal with files(html,xml..)
- ✓Ability to deal with databases.

Nodejs Server Installation

Installing Node.js



Running Windows power Shell (cmd)

```
C:\Users\ITI>node --version
v16.13.1

C:\Users\ITI>npm --version
8.1.2
```

Node.js REPL

Node.js comes with virtual environment called **REPL** (Node shell). REPL stands for **R**ead-**E**val-**P**rint-**L**oop. It is a quick and easy way to test simple Node.js/JavaScript code.

Node.js REPL important commands

Command	Description
.save filename	Save current Node REPL session to a file.
.load filename	Load the specified file in the current Node REPL session.
ctrl + c	Terminate the current command.
ctrl + c (twice)	Exit from the REPL.
ctrl + d	Exit from the REPL.
.break	Exit from multiline expression.
.clear	Exit from multiline expression.

Node.js Global Objects

Objects that are available globally on the **global** namespace. We can use those objects immediately on JavaScript file without requiring anything.

- ✓ console
- ✓ __dirname
- ✓ __filename
- ✓ exports
- ✓ module
- ✓ setInterval
- ✓ setTimeout
- ✓ process

Node.js Global Objects

In browsers, the top-level scope is the global scope. This means that within the browser **var X** will define a new global variable. In Node.js this is different. The top-level scope is not the **global** scope; **var X** inside a Node.js module will be **local** to that module.

```
console.log("Hello") === global.console.log("Hello"); //true

let instructorName="Eman"; //local to the module
course="NodeJs"; //added to global object

console.log(global.course); //"NodeJs"
console.log(global.instructorName); //undefined
```

Node.js Global Objects

One important object that is available globally is **process** object. It contains functionality that allows dealing with current process instance (get environment information, environment variables, communicate with terminal or parent processes) .
By using process object , any information can be available when application starts.

```
process.env ;// all environment information
process.env.lang ;// 'en_US.UTF-8'
process.exit(-1) //exit current process
```

- ✓Organize code into reusable code.
- ✓Dealing with the requests over the internet .
- ✓Ability to accept request and send responses back to the client.
- ✓Ability to deal with files(html,xml..)
- ✓Ability to deal with databases.

NodeJs Modules

Node.js Modules

Before writing Node.js applications, you must learn about Node.js **modules** and **packages**. Modules and packages are the building blocks for breaking down your application into smaller pieces. Every JavaScript file used in Node.js is itself a module

"Modules are the basic building block for constructing Node.js applications."

Node.js's module implementation is strongly inspired by, but not identical to, the **CommonJS** module specification.

Node.js does not support ES2015 modules but **Babel** supports transpiling ES2015 modules to Node.js modules. Starting with version **8.5.0**, Node.js supports ES modules natively, behind a command line option.

Defining a module

There are essentially two elements to interact with modules: **require** and **exports**.

The **require** function searches for modules, loading the module definition into the Node.js runtime, and makes its functions available. Anything assigned to a field of the **exports** object is available to other pieces of code, and everything else is hidden. The exports object is what's returned by **require('./module')**.

Defining a module

```
let ProjectId=30;
let projectDuration="20 Weeks";
exports.projectDuration= projectDuration;
exports.projectSupervisor="Eman fathi";
exports.projectData=function(){ return {ProjectId} }
```

Module1.js

```
let module1Values=require("./module1");
let InstructorName=module1Values.projectSupervisor;
let ProjectNumber=module1Values.projectDuration;
module1Values.projectData();
```

Module2.js

node.js™

+

npm

npm – Node Package Manager

Its' purpose is publishing and distributing Node.js packages over the Internet using a simple command-line interface. With npm, you can quickly **find** packages to serve specific purposes, **download** them, **install** them, and **manage** packages you've already installed.

The **npm package** defines a **package format** for Node.js largely based on the CommonJS package specification. It uses the same **package.json** file that's supported natively by Node.js, but with additional fields to build in additional functionality.

Website :https://npmjs.org

NPM, Inc. [US] | https://www.npmjs.com

Not Parents' Money

npm Enterprise Features Pricing Docs Support

npm

Search packages

log in or sign up

Build amazing things

npm is the package manager for JavaScript and the world's largest software registry. Discover packages of reusable code — and assemble them in powerful new ways.

Sign up

Installing an npm package

In Dependencies:

npm install express –save

In DevDependencies:

npm install express –save-dev

The named module is installed in node_modules in the current directory.

Globally:

If you instead want to install a module globally, add the -g option:

npm install noedmon -g

The npm package format – Package.json

An **npm** package is a directory structure with a **package.json** file describing the package.

A basic package.json file is as follows:

```
{  "name": "packageName",
   "version": "1.0",
   "main": "mainModuleName.js",
   "scripts": {
              "start": "node mainModuleName.js",
              }
}
```

The file is in JSON format, which, as a JavaScript programmer, you should be familiar with.

Module identifiers and path names

There are three types of module identifiers: relative, top-level and core

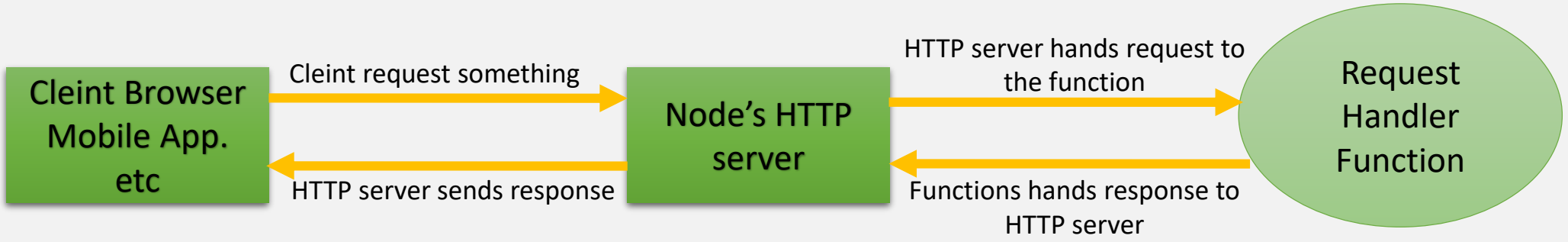
- ✓ **Relative module identifiers:** These begin with ./ or ../ That is, a module identifier beginning with ./ is looked for in the current directory; whereas, one starting with ../ is looked for in the parent directory.
- ✓ **Top-level module identifiers:** These begin with none of those strings and are just the module name. These must be stored in a node_modules directory, and the Node.js runtime has a nicely flexible algorithm for locating the correct node_modules directory.
- ✓ **Core modules :** exists where nodejs installed

- ✓Organize code into reusable code.
- ✓Dealing with the requests over the internet .
- ✓Ability to accept request and send responses back to the client.
- ✓Ability to deal with files(html,xml..)
- ✓Ability to deal with databases.

HTTP server applications

When creating web application in Node.js (**web server**), it contains only one javascript function for entire application. This function listens to a web browser's requests.When a request comes in, this function will look at the request and determine how to respond.

The JavaScript function that processes browser requests in your application is called a **request handler**.



HTTP server applications

In code, it's a function that takes two arguments: an object that represents the request and an object that represents the response. Every Node.js application is just like this: it's a single request handler function responding to request.

The problem is that the Node.js APIs can get complex. **Express** was born to make it easier to write web applications with Node.js.

```
var http = require('http');
http.createServer((req, res) => {
        res.writeHead(200, {'Content-Type': 'text/plain'});
        res.end('Hello, World!\n');
}).listen(8124);
console.log('Server running at http://127.0.0.1:8124');
```

- ✓Organize code into reusable code.
- ✓Dealing with the requests over the internet .
- ✓Ability to accept request and send responses back to the client.
- ✓Ability to deal with files(html,xml..)
- ✓**Ability to deal with databases.**

File system

Javscript can not open files or write on file, in nodejs javascript use file packge to send file to any incoming request.

```
var fs = require('fs');
fs.readFile(path.join(__dirname,"home.html"),(error,file)=>{
    if(!error)
    {
        res.end(file);
    }
})
```


express
TM
Eman Fathi
https://expressjs.com

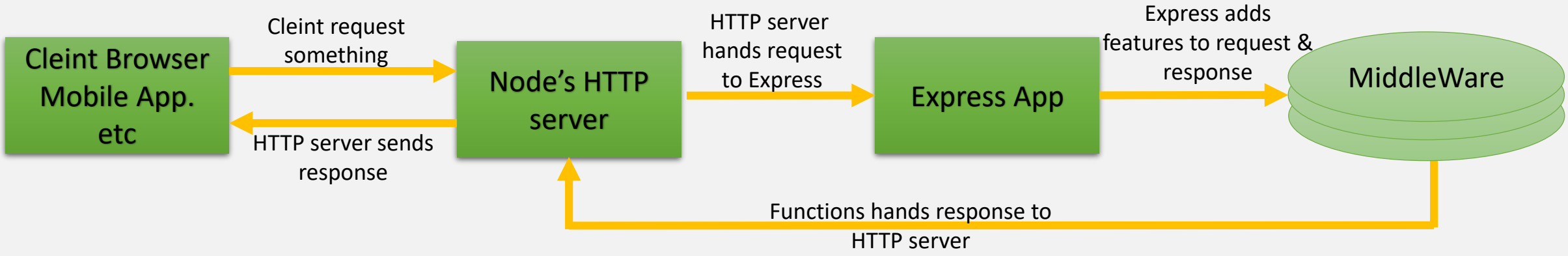
Getting started with Express

Express is perhaps the most popular Node.js web app framework. It's so popular that it's part of the MEAN Stack acronym. **MEAN** refers to **M**ongoDB, **E**xpressJS, **A**ngularJS, and **N**ode.js

What is Express

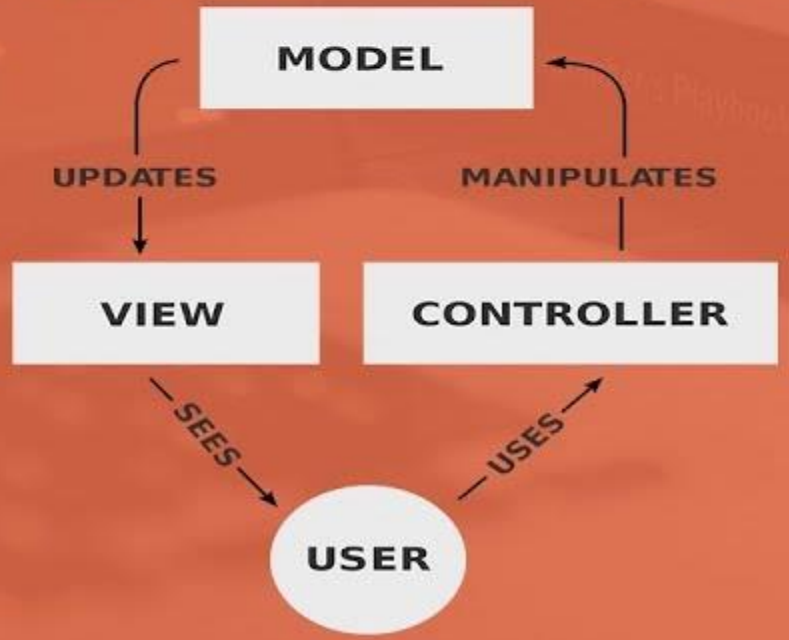
Express is a relatively small framework that sits on top of Node.js's web server functionality to simplify its APIs and add helpful new features.

- ✓It makes it easier to organize your application's functionality with middleware and routing
- ✓It adds helpful utilities to Node.js's HTTP objects
- ✓It facilitates the rendering of dynamic HTML views



What is The MVC?

Model - View - Controller



node JS

&

Express

Foundations of Express

Express is an abstraction layer on top of Node's built-in HTTP server. Express provides four major features

- ✓**Middleware**:where your requests flow through only one function, Express has a middleware stack, which is effectively an array of functions.
- ✓**Routing** : Routing is a lot like middleware, but the functions are called only when you visit a specific URL with a specific HTTP method.
- ✓**Extensions to request and response objects**: Express extends the request and response objects with extra methods and properties for developer convenience.
- ✓ **Views**:Views allow you to dynamically render HTML and Change HTML on the fly.

Middleware

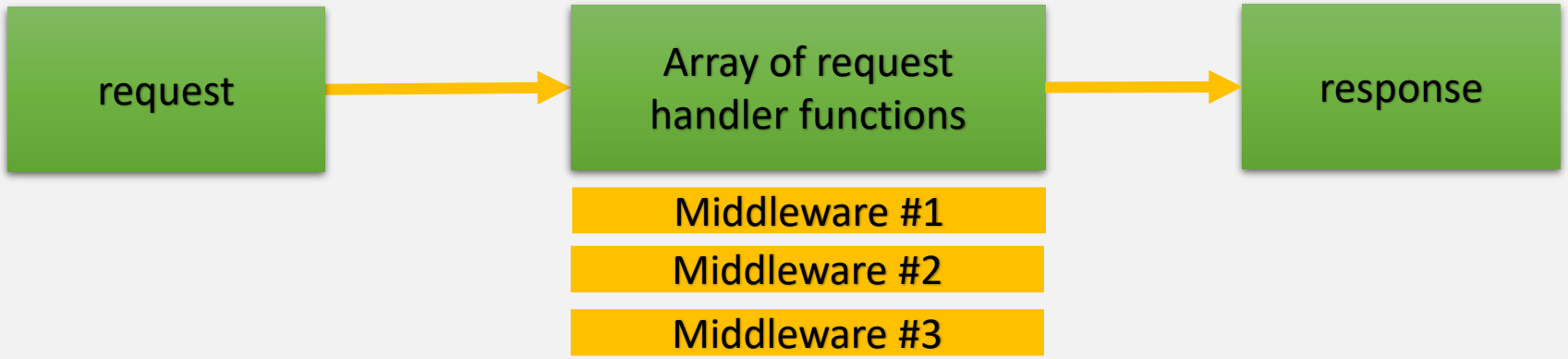
Middleware is very similar to the request handlers in Node.js, but middleware has one important difference: rather than having just one handler, middleware allows for many to happen in sequence.

```
let express=require("express");
let app=express();
//middleware
app.use((request,response)=>{
	console.log(request.url, request.method);
	response.end("Hello first Express app");
});
app.listen(8080,function(){
console.log("app starting listen to post 8080")
});
```

With middleware, rather than having your request pass through one function you write, it passes through an array of functions you write called a middleware stack.

In Node.js , request and response object are passed to only **one** function. But in Express these objects are passed through an **array** of functions called the middleware stack.

Express will start at the first function in the stack and continue in order down the stack.



Every function in this stack takes three arguments. The first two are request and response from before. They're given to you by Node, The third argument to each of these functions is itself a function, conventionally called next. When next is called, Express will go on to the next function in the stack.

Eventually, one of these functions in the stack must call res.end, which will end the request.

```
let express=require("express");
let app=express();
//middlewares
app.use((request,response,next)=>{
        console.log(request.url,request.method);
        next();
});
app.use((request,response)=>{
        response.writeHead(200,"text/plain");
        response.end("passing form last middleware")
});
app.listen(8080);
```

Error handling Middleware

These middleware functions take **four** arguments instead of two or three. The first one is the error (the argument passed into next), and the remainder are the three from before: req, res, and next.
While not enforced, error-handling middleware is conventionally placed at the end of your middleware stack, after all the normal middleware has been added.
If no errors happen, it'll be as if the error-handling middleware never existed. To reiterate more precisely, "no errors" means "next was never called with any arguments." If an error does happen, then Express will skip over all other middleware until the first error-handling middleware in the stack.

```
app.use((request,response,next)=>{
        if(condition)
        {
                next( new Error("Bad data"));
        }
        else
        {
                next();
        }
});

app.use((request,response)=>{
        response.writeHead(200,"text/plian");
        response.end("passing form last middleware")
});

app.use((err,req,res,next)=>{
        //handling Error
});
```

Built-in Middlewares

- ✓ Morgan module : containing info about loggers
- ✓ Body-parse module :handles parsing HTTP request bodies
- ✓ Cookie-parser module : used to parse http cookies
- ✓ static file Express Middleware : web server configured to serve the asset files in the public directory.

Built-in Static File Middleware

express.static is a function that returns a middleware function. It takes one argument: the path to the folder you'll be using for static files(path.join).
If the file exists at the path, it will send it. If not, it will call next and continue on to the next middleware in the stack.

```
let express = require("express");
let path = require("path");
let app = express();
let staticPath = path.join(__dirname,"folderName");
app.use(express.static(staticPath));
```

Routing

Routing is a way to map requests to specific handlers depending on their URL and HTTP verbs (GET ,POST ,PUT and DELETE).
They work just like middleware; it's a matter of when they're called.

```
app.get("/about",function(req,res,next){
    res.end("welcome to about page");
});

app.post("/register",function(req,res){
   res.end(" welcome to register page ");
});
```

Grabbing parameters to routes

These routes can get smarter. In addition to matching fixed routes, they can match more complex ones (imagine a regular expression or more complicated parsing)

```
app.get("/hello/:who",(req,res)=>{
    res.send(req.params.who);
});
```

It's no coincidence that this who is the specified part in the first route. Express will pull the value from the incoming URL and set it to the name you specify.

visiting localhost:8080/hello/earth for the following message: Hello, earth. Note that this won't work if you add something after the slash. For example, localhost:8080/hello/entire/earth will give a 404 error.

Grabbing query arguments

Another common way to dynamically pass information in URLs is to use query strings.

```
app.get("/login",function(req,res){
        res.end(" welcome to register page "+req.query.name);
});
```

Using Routers

Express 4 added **routers**, a feature to help ease these growing pains. **Router** allow s you to chunk your big app into numerous mini-apps that you can later put together.

"A router is an isolated instance of middleware and routes."

```
let userRouter=express.Router();

userRouter.use((req,res,next)=>{
        console.log("authorized");
        next();
}).get("/login",(req,res)=>{
        res.send("get");
}).post("/login",(req,res)=>{
        res.send("post");
});
app.use("/users",userRouter);
```

Views

Views are dynamically generated html. A number of different view engines are available. There's **EJS** (Embedded JavaScript), Handlebars, Pug, and more. All of these have one thing in common: at the end of the day, they spit out HTML.

```
npm install ejs --save.
```

Then:

```
app.set("view  engine","ejs");
app.set("views",path.join(__dirname,"/views"));
```

Simple View rendering

Main app.js file

```
app.use("/welcome",(req,res)=>{res.render("welcome",{message:" ITI "});
});
```

Welcome.ejs file

```
<html>
<head> Welcome Page</head>
<body>
       welcome every one to <%=message %> website
</body>
</html>
```

Simple View rendering

Main app.js file

```
app.use("/welcome",(req,res)=>{ res.locals.message="iti";
                                res.render("welcome");
});
```

Welcome.ejs file

```
<html>
<head> Welcome Page</head>
<body>
      welcome every one to <%=message %> website
</body>
</html>
```

User Sessions

Sessions are used to keep user authenticated on website. Express session package is the best choice

```
npm install express-session --save.
```

```
let session=require("express-session");
app.use(session({
      secret:"secret key",
      resave:false,
      saveUninitialized:true
}));
```

Thank You!