NODE JS

Aruna S
Department of
Computer Science and Engineering

NODE JS

NodeJS Introduction

S. Aruna
Department of Computer Science and Engineering

NODE JS

NodeJS Introduction

- Node.js is an open source, cross-platform runtime environment built on Google Chrome's JavaScript Engine (V8 Engine).
- Node.js is a platform built on chrome’s JavaScript runtime for easily building fast and scalable network applications.
- Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.
- Node.js is open source, completely free, and used by thousands of developers around the world.

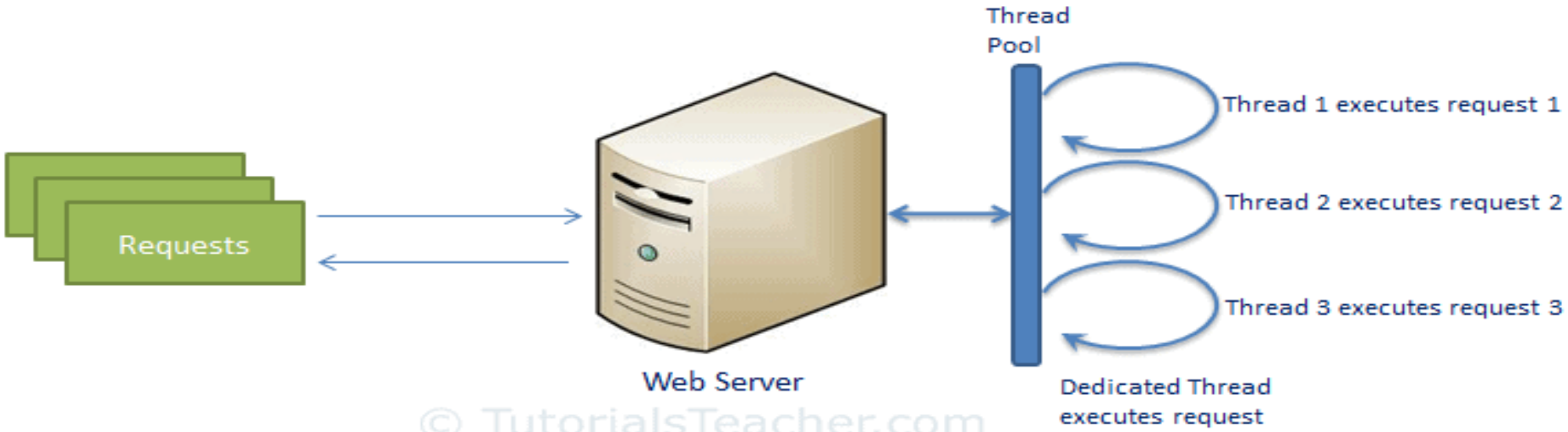
NodeJS Introduction

- Node.js applications are written in JavaScript, and can be run within the Node.js runtime on OS X, Microsoft Windows, and Linux.
- Node.js also provides a rich library of various JavaScript modules which simplifies the development of web applications using Node.js to a great extent.

Node.js = Runtime Environment + JavaScript Library

Traditional Web Server Model

In the traditional web server model, each request is handled by a dedicated thread from the thread pool. If no thread is available in the thread pool at any point of time then the request waits till the next available thread. Dedicated thread executes a particular request and does not return to thread pool until it completes the execution and returns a response.



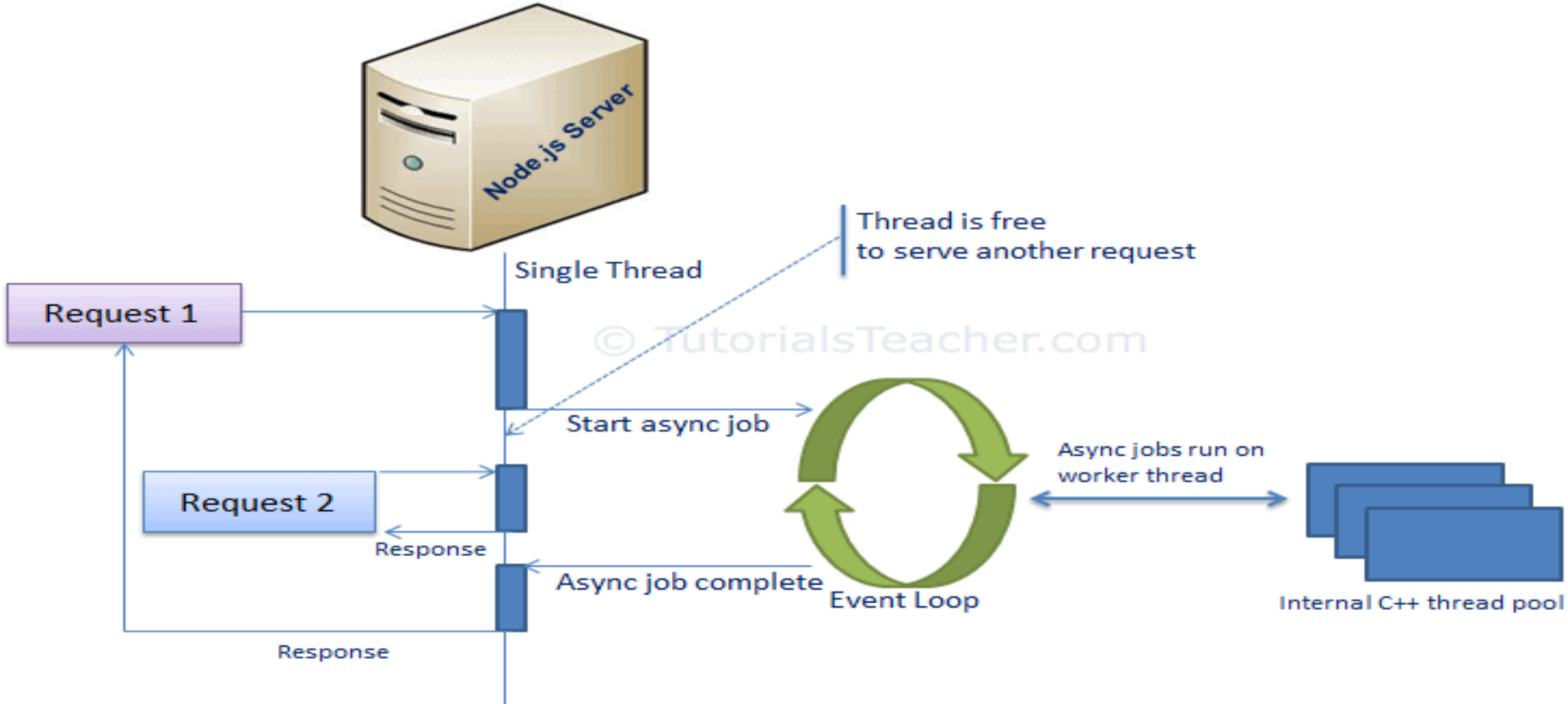
Node.js Process Model

Node.js runs in a single process and the application code runs in a single thread and thereby needs less resources than other platforms. All the user requests to your web application will be handled by a single thread and all the I/O work or long running job is performed asynchronously for a particular request. So, this single thread doesn't have to wait for the request to complete and is free to handle the next request. When asynchronous I/O work completes then it processes the request further and sends the response.

NODE JS

Mono-Thread Style

An event loop is constantly watching for the events to be raised for an asynchronous job and executing callback function when the job completes. Internally, Node.js uses libev for the event loop which in turn uses internal C++ thread pool to provide asynchronous I/O.



NODE JS

V8 Engine and C++ Bindings

- Chrome needs to run some Javascript for a particular Web page, it doesn't run the JavaScript itself. It uses the V8 engine to get that done so it passes the code to V8 and it gets the results back. Same case with Node also to run a Javascript code.
- V8 is written in C++ . Chrome and Node are largely written in C++ because they both provide bindings when they're instantiating the V8 engine.
- This facilitates to create their own JavaScript runtime with interesting and novel features.

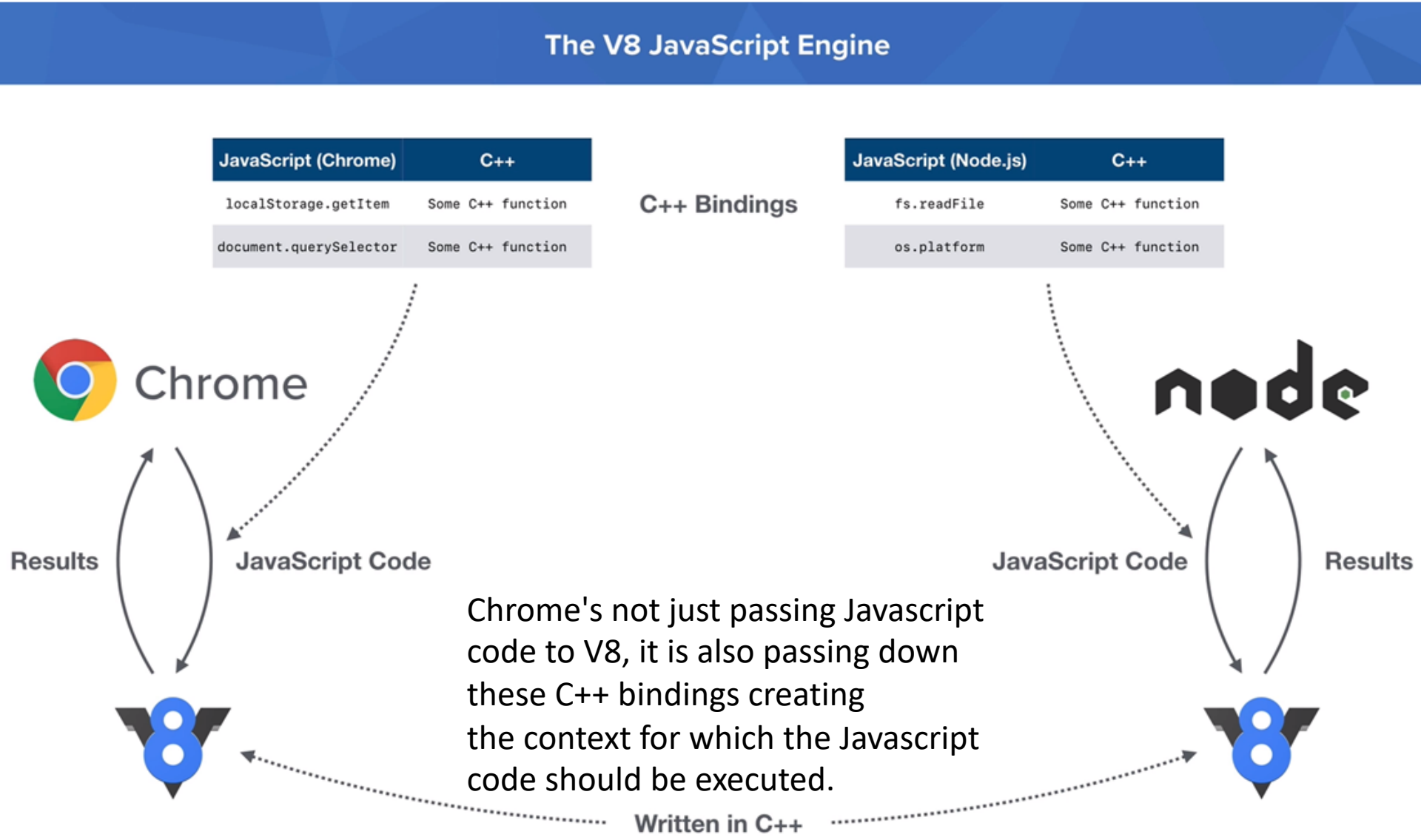
Example Instance:

Chrome to interact with the DOM when the DOM isn't part of JavaScript.

Node to interact with file system when the file system isn't part of JavaScript

NODE JS

V8 JavaScript Engine



Chrome's not just passing Javascript code to V8, it is also passing down these C++ bindings creating the context for which the Javascript code should be executed.

NODE JS

Features of Node.js

- Asynchronous and Event Driven
- Non Blocking I/O
- Very Fast
- Single Threaded but Highly Scalable
- No Buffering
- License

NODE JS

Features of Node.js

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

Uses a single threaded model with event looping. Event mechanism helps the server to respond in a non-blocking way and makes the server highly scalable as opposed to traditional servers.

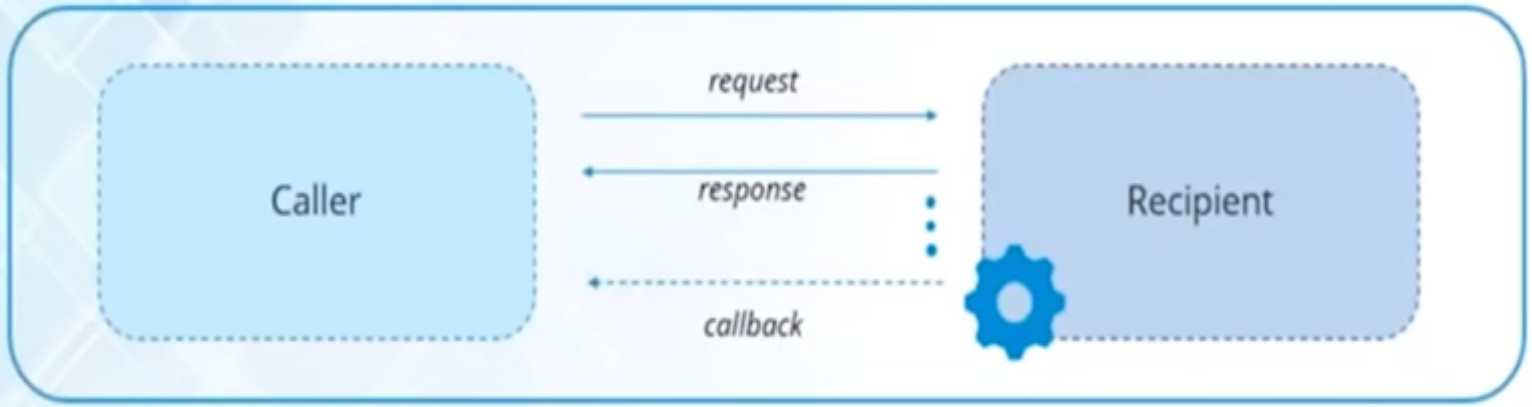
No Buffering

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

NODE JS

Asynchronous and Event Driven

Asynchronous and Event Driven :



- When request is made to server, instead of waiting for the request to complete, server continues to process other requests
- When request processing completes, the response is sent to caller using callback mechanism

NODE JS

Callback Functions

- 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.

NODE JS

Blocking and Non- Blocking I/O

BLOCKING I/O

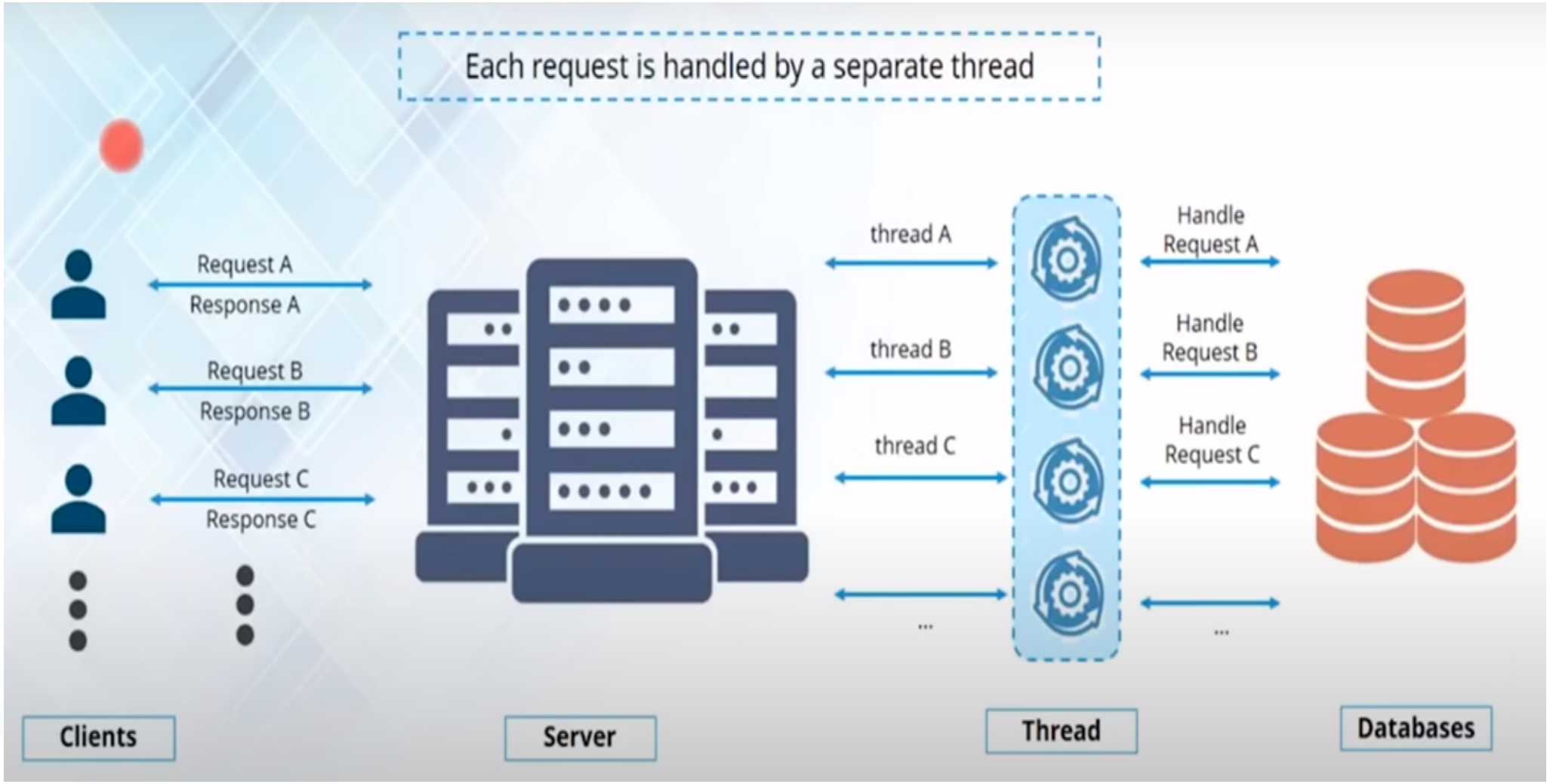
```
var fs = require('fs');
var data = fs.readFileSync('test.txt');
console.log(data.toString());
console.log('End here');
```

```
var fs = require('fs');
fs.readFile('test.txt',function(err,data){
    if(err)
    {
        console.log(err);
    }
    setTimeout(()=>{
        console.log("PES University. Display after 2 seconds")
    },2000);
});
console.log('start here');
```

NON-BLOCKING I/O

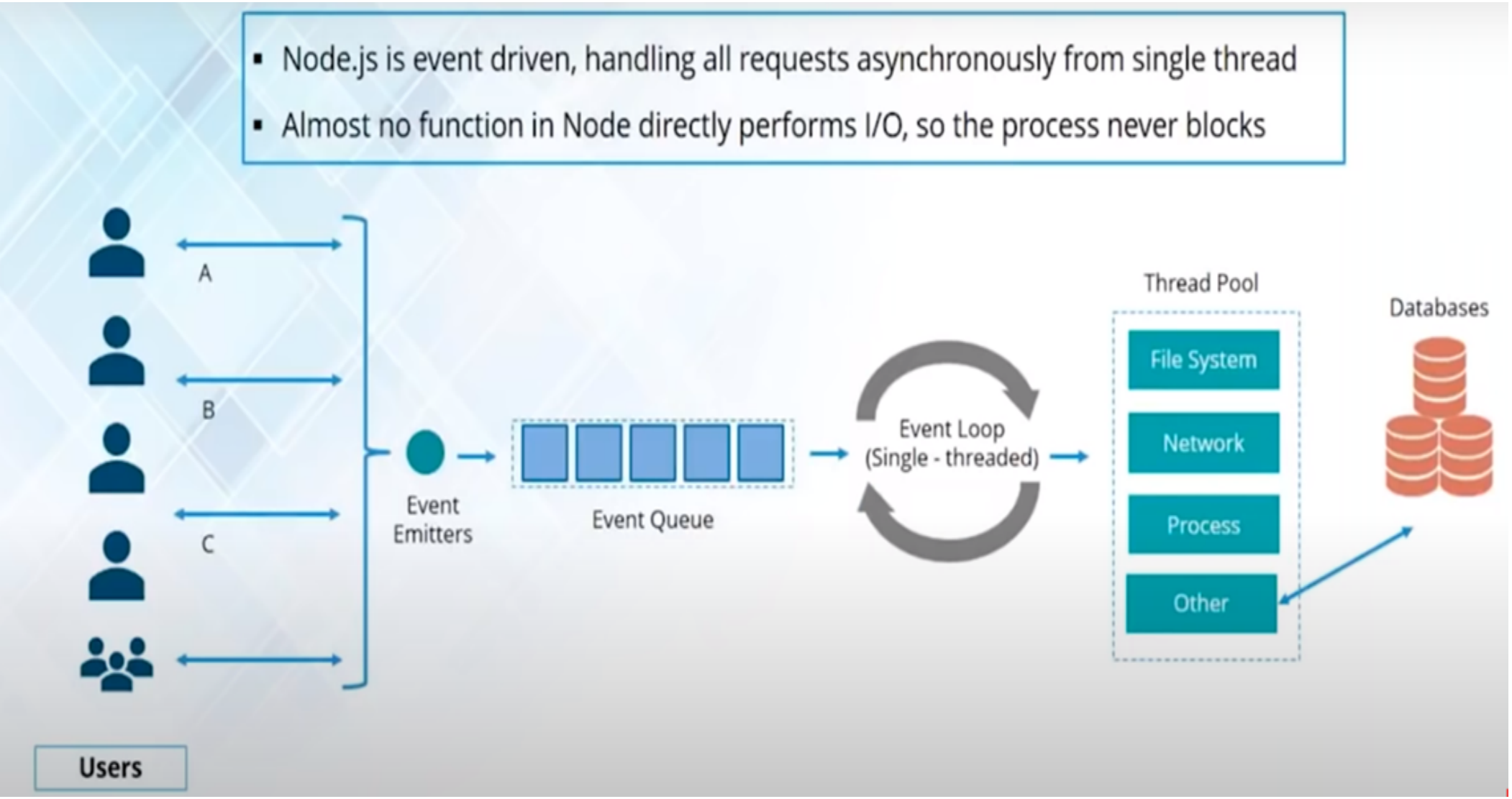
NODE JS

Multi Threaded Model



NODE JS

Single Threaded Model



NODE JS

Multi Threaded vs Asynchronous Event Driven Model

Multi-Threaded	Asynchronous Event-driven
Lock application / request with listener-workers threads	Only one thread, which repeatedly fetches an event
Using incoming-request model	Using queue and then processes it
Multithreaded server might block the request which might involve multiple events	Manually saves state and then goes on to process the next event
Using context switching	No contention and no context switches
Using multithreading environments where listener and workers threads are used frequently to take an incoming-request lock	Using asynchronous I/O facilities (callbacks, not poll/select or O_NONBLOCK) environments

Applications of Node.js

- I/O bound Applications
- Data Streaming Applications
- Data Intensive Real-time Applications (DIRT)
- JSON APIs based Applications
- Single Page Applications
- Not for CPU intensive applications.

NODE JS

Node JS Success Stories

Nexflix used JavaScript and NodeJS to transform their website into a single page application.

NETFLIX

PayPal team developed the application simultaneously using Java and Javascript. The JavaScript team build the product both faster and more efficiently.

PayPal

Uber has built its massive driver / rider matching system on Node.js Distributed Web Architecture.

UBER

Node enables to build quality applications, deploy new features, write unit and integration tests easily.

GoDaddy.COM

When LinkedIn went to rebuild their Mobile application they used Node.js for their Mobile application server which acts as a REST endpoint for Mobile devices.

LinkedIn

They had two primary requirements: first to make the application as real time as possible. Second was to orchestrate a huge number of eBay-specific services.

ebay

UBER Story

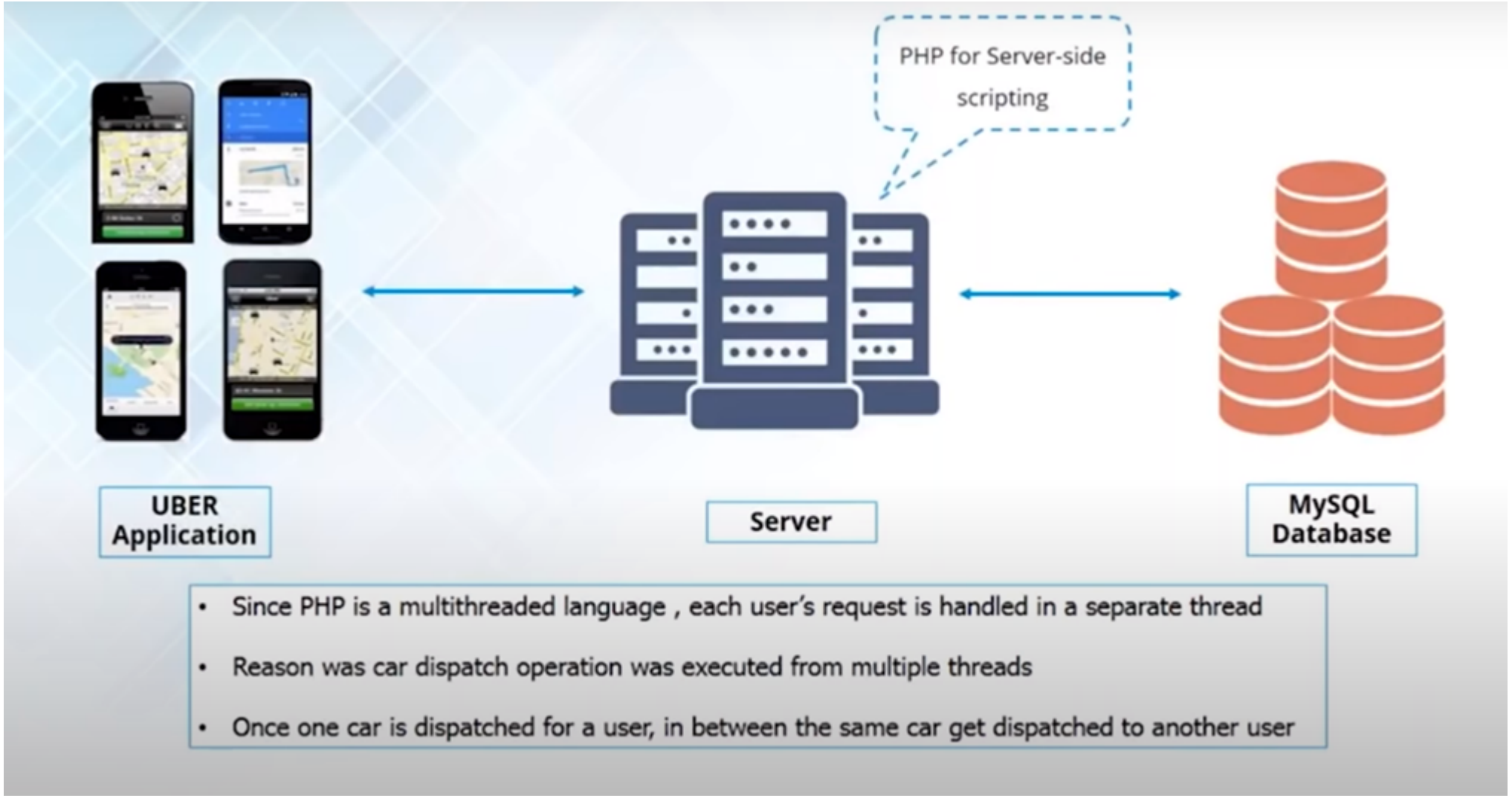
- Customers interact with Uber applications and generating request to book a cab.
- Requests are sent to the Uber server to check in the geospatial databases for the cab.
- Server send back the card details or driver details back to the customer in form of response.

Disadvantage of Multi Threaded model in this case:

- Every request is assigned a new thread from thread pool and it gets exhausted. Scalability is very poor
- Whenever a thread is working on a shared resource. It acquires a lock on that resource.

NODE JS

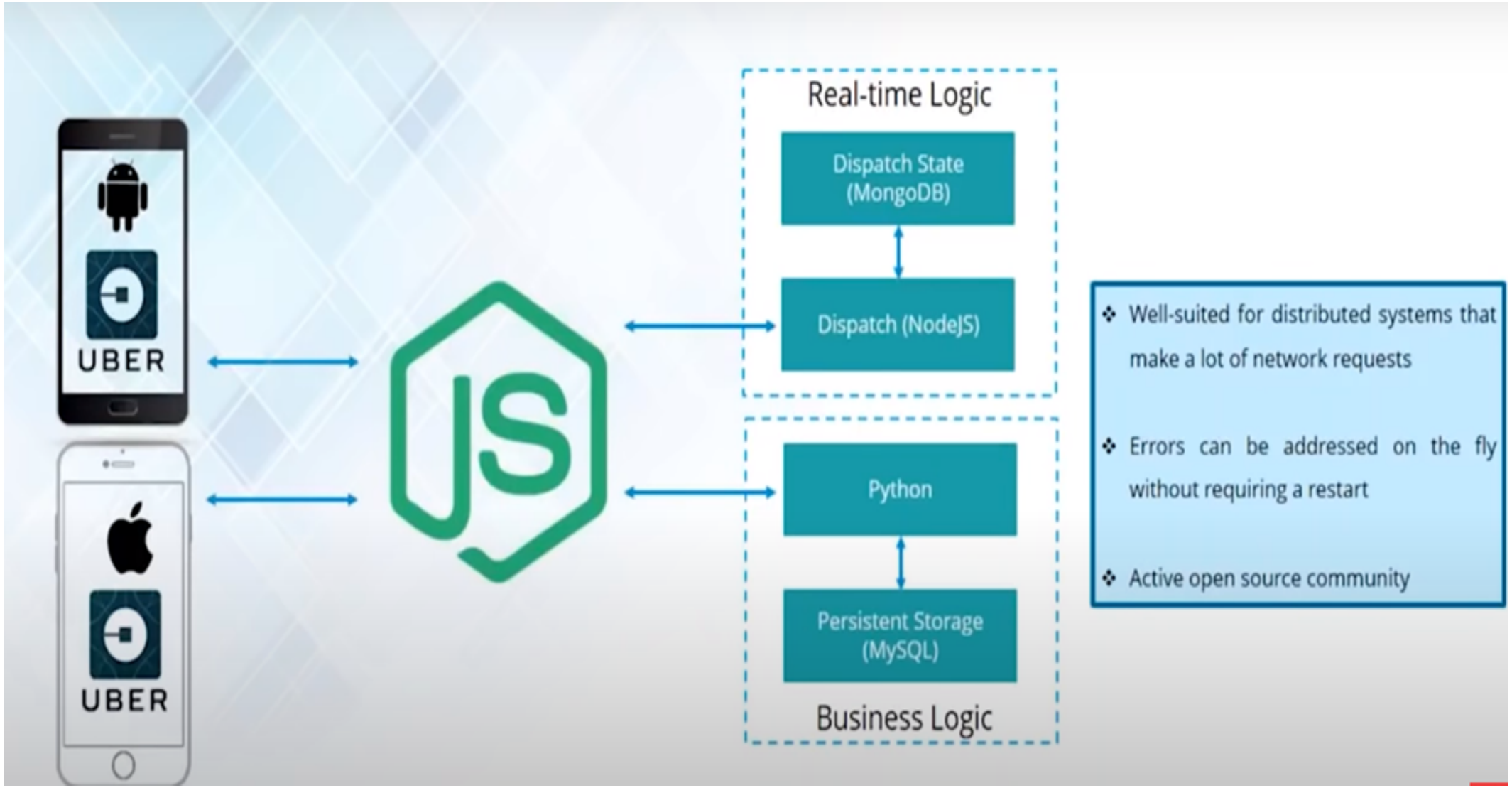
UBER Story – Old Architecture



- Since PHP is a multithreaded language , each user's request is handled in a separate thread
- Reason was car dispatch operation was executed from multiple threads
- Once one car is dispatched for a user, in between the same car get dispatched to another user

NODE JS

UBER Story – New Architecture



NODE JS

UBER Story – Single threaded Model

- User performs an activity or event is generated, each new request is taken as an event.
- Event Emitter emit those events and then those events reside inside the event queue in the server.
- Events are executed using event Loops,which is a single thread mechanism
- A worker thread present inside the thread pool is assigned for each request.
- Only one thread in this event Loop will be handling the events directly and process will never get Blocked.

NODE JS

Node JS installation

- Go to https://nodejs.org/en/
- Look for the Latest Stable Version and download it
- After Installing, Verify the installation by giving the command as

```
C:\Users\DELL>node -v
v12.18.3
```

- Install a editor like Visual Code, Sublime Text or any suitable editors for running Node JS Applications

Node JS Modules

- https://nodejs.org/dist/latest-v12.x/docs/api/

  To see all the available modules

- Few modules are inbuilt globally available.

  Ex: Console module, Timer Module

- Many modules need to be explicitly included in our application

  Ex: File System module

Such modules need to be required at first in the application

NODE JS

Node JS Timer Module

- This module provides a way for functions to be called later at a given time.
- The Timer object is a global object in Node.js, and it is not necessary to import it

Method	Description
clearImmediate()	Cancels an Immediate object
clearInterval()	Cancels an Interval object
clearTimeout()	Cancels a Timeout object
ref()	Makes the Timeout object active. Will only have an effect if the Timeout.unref() method has been called to make the Timeout object inactive.
setImmediate()	Executes a given function immediately.
setInterval()	Executes a given function at every given milliseconds
setTimeout()	Executes a given function after a given time (in milliseconds)
unref()	Stops the Timeout object from remaining active

NODE JS

NPM and installing modules

- NPM is a package manager for Node.js packages, or modules if you like.
- www.npmjs.com hosts thousands of free packages to download and use.
- The NPM program is installed on your computer when you install Node.js

  A package in Node.js contains all the files you need for a module.

  Modules are JavaScript libraries you can include in your project.

Example:

```
D:\nodejs>npm install validator
```

- validator package is downloaded and installed. NPM creates a folder named "node_modules", where the package will be placed.
- To include a module, use the **require()** function with the name of the module

```
var val = require('validator');
```

File System Module

- Node implements File I/O using simple wrappers around standard POSIX functions.
- The Node File System (fs) module can be imported using the following syntax –

```
const fs = require('fs');
```

Synchronous vs Asynchronous

- Every method in the fs module has synchronous as well as asynchronous forms.
- Asynchronous methods take the last parameter as the completion function callback and the first parameter of the callback function as error.
- It is better to use an asynchronous method instead of a synchronous method, as the former never blocks a program during its execution, whereas the second one does.

NODE JS

Fs module operations

Common use for the File System module:

- Read files
- Create files
- Update files
- Delete files
- Rename files

Syntax

Following is the syntax of the method to open a file in asynchronous mode −

fs.open(path, flags[, mode], callback)

Parameters

Here is the description of the parameters used −

path − This is the string having file name including path.

flags − Flags indicate the behavior of the file to be opened. All possible values have been mentioned below.

mode − It sets the file mode (permission and sticky bits), but only if the file was created. It defaults to 0666, readable and writeable.

callback − This is the callback function which gets two arguments (err, fd).

Flags for read/write operations are − r,r+, w,wx,w+,wx+,a,ax,a+,ax+

Syntax

fs.writeFile(filename, data[, options], callback)

This method will over-write the file if the file already exists. If you want to write into an existing file then you should use another method available.

Parameters

path – This is the string having the file name including path.
data – This is the String or Buffer to be written into the file.
options – The third parameter is an object which will hold {encoding, mode, flag}. By default. encoding is utf8, mode is octal value 0666. and flag is 'w'
callback – This is the callback function which gets a single parameter err that returns an error in case of any writing error.

Syntax

fs.read(fd, buffer, offset, length, position, callback)

This method will use file descriptor to read the file. If you want to read the file directly using the file name, then you should use another method available.

Parameters

fd – This is the file descriptor returned by fs.open().
buffer – This is the buffer that the data will be written to.
offset – This is the offset in the buffer to start writing at.
length – This is an integer specifying the number of bytes to read.
position – This is an integer specifying where to begin reading from in the file. If position is null, data will be read from the current file position.
callback – This is the callback function which gets the three arguments, (err, bytesRead, buffer).

Unlinking a File

Use fs.unlink() method to delete an existing file.

```
fs.unlink(path, callback);
```

Closing a File

```
fs.close(fd, callback)
```

fd – This is the file descriptor returned by file fs.open() method.
callback – This is the callback function No arguments other than a possible exception are given to the completion callback.

Truncate a File

```
fs.truncate(fd, len, callback)
```

fd – This is the file descriptor returned by fs.open().
len – This is the length of the file after which the file will be truncated.
callback – This is the callback function No arguments other than a possible exception are given to the completion callback.

NODE JS

Fs Module – Other Important Methods

Method	Description
fs.readFile(fileName [,options], callback)	Reads existing file.
fs.writeFile(filename, data[, options], callback)	Writes to the file. If file exists then overwrite the content otherwise creates new file.
fs.open(path, flags[, mode], callback)	Opens file for reading or writing.
fs.rename(oldPath, newPath, callback)	Renames an existing file.
fs.chown(path, uid, gid, callback)	Asynchronous chown.
fs.stat(path, callback)	Returns fs.stat object which includes important file statistics.
fs.link(srcpath, dstpath, callback)	Links file asynchronously.
fs.symlink(destination, path[, type], callback)	Symlink asynchronously.
fs.rmdir(path, callback)	Renames an existing directory.
fs.mkdir(path[, mode], callback)	Creates a new directory.
fs.readdir(path, callback)	Reads the content of the specified directory.
fs.utimes(path, atime, mtime, callback)	Changes the timestamp of the file.
fs.exists(path, callback)	Determines whether the specified file exists or not.
fs.access(path[, mode], callback)	Tests a user's permissions for the specified file.
fs.appendFile(file, data[, options], callback)	Appends new content to the existing file.

NODE JS

Importing your own modules

- The module.exports is a special object which is included in every JavaScript file in the Node.js application by default.
- The module is a variable that represents the current module, and exports is an object that will be exposed as a module.
- So, whatever you assign to module.exports will be exposed as a module. It can be
  - Export Literals
  - Export Objects
  - Export Functions
  - Export Function as a class

NODE JS

Importing your own modules

- Export Literals

if you assign a string literal then it will expose that string literal as a module.

Eg –

```
module.exports = 'Hello world';  // in message.js

//import this message module and use it in app.js

var msg = require('./Messages.js');

console.log(msg);
```

We must specify ./ as a path of root folder to import a local module. However, you do not need to specify the path to import Node.js core modules or NPM modules in the require() function.

Importing your own modules

- Export Object

The exports is an object. So, you can attach properties or methods to it.

Eg –

```
 exports.SimpleMessage = 'Hello world';
//or
module.exports.SimpleMessage = 'Hello world'; // in Messages.js

//import and use this module

var msg = require('./Messages.js');
console.log(msg.SimpleMessage);  // in app.js
```

Importing your own modules

- Export Functions

```
exports.myadd = function add(a,b){
        return a+b;
}
exports.mysub = function sub(a,b){
        return a-b;                              // in function.js file
}

console.log("Sum of 10 and 50 is :" +lib.myadd(10,50))
console.log("Difference of 50 and 10 is :" +lib.mysub(50,10))    // in app.js
```

Importing your own modules

- Export Function as a Class

In JavaScript, a function can be treated like a class.

```
module.exports = function (firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.fullName = function () {
        return this.firstName + ' ' + this.lastName;     // in Person.js
    }
}
var person = require('./Person.js');
//we have created a person object using the new keyword
var person1 = new person('James', 'Bond');
console.log(person1.fullName());   //in app.js
```

NODE JS

Importing your own modules

You can create your own modules, and easily include them in your applications. The following example creates a module that returns a date and

```
exports.myDateTime = function () {
    return Date();
  };
```

Use the exports keyword to make properties and methods available outside the module file.

```
var date = require('./myfirstmodule.js');
console.log(date.myDateTime());
```

```
PS C:\Users\DELL\Desktop\WTII\Node Examples\notes-app> node app.js
Sat Sep 12 2020 12:03:34 GMT+0530 (India Standard Time)
```

NODE JS

Package.json file

- All npm packages contain a file, usually in the project root, called package.json
- This file holds various metadata relevant to the project.
- It is used to give information to npm that allows it to identify the project as well as handle the project's dependencies.
- It can also contain other metadata such as a project description, the version of the project in a particular distribution, license information, even configuration data - all of which can be vital to both npm and to the end users of the package.
- The package.json file is normally located at the root directory of a Node.js project.

NODE JS

A Sample Package.json file

```
{
    "name": "sample",
    "version": "1.0.0",
    "description": "Learning Express",
    "main": "index.js",
    "dependencies": {
      "body-parser": "^1.19.0",
      "builtin-modules": "^3.1.0",
      "express": "^4.17.1",
      "mongodb": "^3.6.1",
      "npm": "^6.14.6"
    },
    "devDependencies": {},
    "scripts": {
      "test": "hi"
    },
    "author": "Aruna",
    "license": "ISC"
  }
```

NODE JS

CHALK and nodemon Module

https://www.npmjs.com/package/chalk

- The Chalk Module Can Be Used With The Console's String Interpolation in Node. Js.
- The Chalk module allows you to add styles to your terminal output.

https://www.npmjs.com/package/nodemon

- **Nodemon** is a utility that will monitor for any changes in your source and automatically restart your server.
- Just use **nodemon** instead of node to run your code
- //npm start

NODE JS

Web Server

- A Web Server is a software application which handles HTTP requests sent by the HTTP client, like web browsers, and returns web pages in response to the clients.
- Web servers usually deliver html documents along with images, style sheets, and scripts.
- Most of the web servers support server-side scripts, using scripting languages or redirecting the task to an application server which retrieves data from a database and performs complex logic and then sends a result to the HTTP client through the Web server.
- Apache web server is one of the most commonly used web servers. It is an open source project.

NODE JS

Web Application Architecture

A Web application is usually divided into four layers −

Client − This layer consists of web browsers, mobile browsers or applications which can make HTTP requests to the web server.

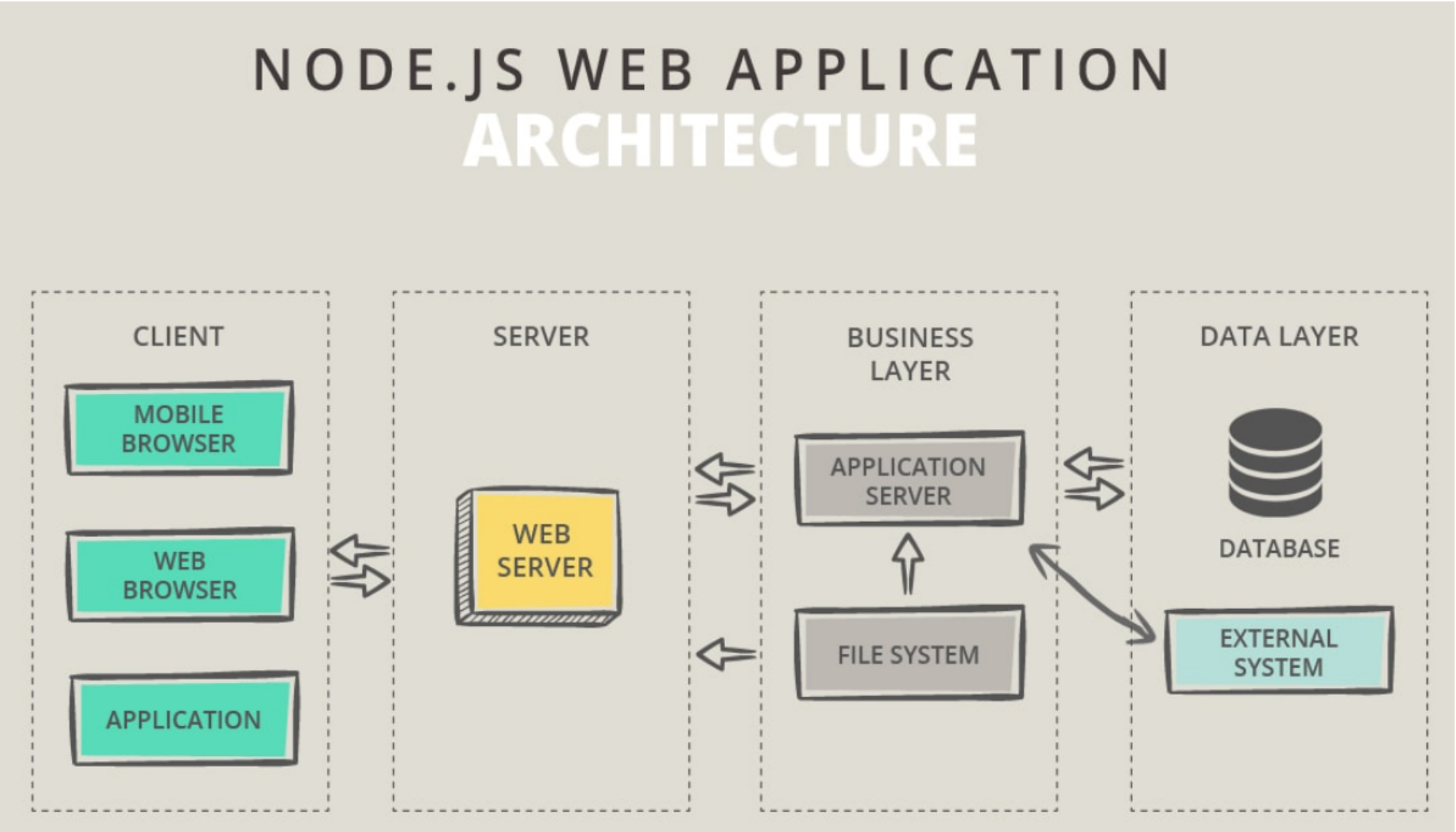
Server − This layer has the Web server which can intercept the requests made by the clients and pass them the response.

Business − This layer contains the application server which is utilized by the web server to do the required processing. This layer interacts with the data layer via the database or some external programs.

Data − This layer contains the databases or any other source of data.

NODE JS

Web Application Architecture



HTTP Module – Web Server using Node

The components of a Node.js application.

Import required modules – We use the **require** directive to load Node.js modules.

Create server – A server which will listen to client's requests similar to Apache HTTP Server. Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).

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.

Creating Node.js Application

Step 1 - Import Required Module

We use the **require** directive to load the http module and store the returned HTTP instance into an http variable as follows

```
var http = require('http');
```

Step 2 - Create Server

- We use the created http instance and call **http.createServer()** method to create a server instance.
- Then we bind it at port 8088 using the **listen** method associated with the server instance.

```
http.createServer(function (request, response) {
    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.end('Hello PES University\n');
 }).listen(8088);
  console.log('Server running at http://127.0.0.1:8088/')
```

Step 3 - Testing Request & Response

$node app.js

Verify the Output. Server has started.

Server running at http://127.0.0.1:8088/

NODE JS

URL Module

The URL module splits up a web address into readable parts.
To include the URL module, use the require() method:

```
var url = require('url');
```

Parse an address with the url.parse() method, and it will return a URL object with each part of the address as properties:

Node Examples > JS app.js > ...

```
1   var url = require('url');
2   var adr = 'http://localhost:8080/pesu.htm?year=2020&month=September';
3   var q = url.parse(adr, true);
4
5   console.log(q.host); //returns 'localhost:8080'
6   console.log(q.pathname); //returns '/pesu.htm'
7   console.log(q.search); //returns '?year=2020&month=September'
8
9   var qdata = q.query; //returns an object: { year: 2020, month: 'september' }
10  console.log(qdata.month); //returns 'september'
```

URL Module

- req.url will look like /pesu.htm?year=2020&month=October. This is the part that is in the URL bar of the browser.
- Next, it gets passed to url.parse which parses out the various elements of the URL (NOTE: the second parameter is a boolean stating whether the method should parse the query string, so we set it to true).
- Finally, we access the .query property, which returns us a nice, friendly JavaScript object with our query string data.

The url.parse() method returns an object which have many key value pairs one of which is the query object. Some other handy information returned by the method include host, pathname, search keys.

NODE JS

Web Client Using Node

A web client can be created using **http** module. A Screenshot of the example is below

```
var http = require('http');
var options = {
   host: 'localhost',
   port: '8081',
   path: '/index.htm'
};
var callback = function(response) {
   var body = '';
   response.on('data', function(data) {
      body += data;
   });

   response.on('end', function() {
      console.log(body);
   });
}
var req = http.request(options, callback);
req.end();
```

Read & Write JSON file Using Nodejs

Reading a JSON file:

Method 1: Using require method: The simplest method to read a JSON file is to require it in a node.js file using require() method.

Syntax:

```
const data = require('path/to/file/filename');
```

NODE JS

Read & Write JSON file Using Nodejs

Users.json

```
{
        "name": "John",
        "age": 21,
        "language": ["JavaScript", "PHP", "Python"]
}
```

Index.js file

```
// Requiring users file
const users = require("./users");
console.log(users);
```

Read & Write JSON file Using Nodejs

Method 2: Using the fs module: We can also use node.js fs module to read a file. The fs module returns a file content in string format so we need to convert it into JSON format by using JSON.parse() in-built method

```
const fs = require("fs");
// Read users.json file
fs.readFile("users.json", function(err, data) {
        // Check for errors
        if (err) throw err;
        // Converting to JSON
        const users = JSON.parse(data);
                console.log(users); // Print users
});
```

Read & Write JSON file Using Nodejs

Writing to a JSON file:

We can write data into a JSON file by using the node.js fs module. We can use writeFile method to write data into a file.

Syntax:

```
fs.writeFile("filename", data, callback)
```

Eg –

We will add a new user to the existing JSON file, we have created in the previous example. This task will be completed in three steps:

- Read the file using one of the above methods.
- Add the data using .push() method.
- Write the new data to the file using JSON.stringify() method to convert data into string.

NODE JS

Read & Write JSON file Using Nodejs

```
const fs = require("fs");
// STEP 1: Reading JSON file
const users = require("./users");
// Defining new user
let user = {
        name: "New User",
        age: 30,
        language: ["PHP", "Go", "JavaScript"]
};
// STEP 2: Adding new data to users object
users.push(user);
// STEP 3: Writing to a file
fs.writeFile("users.json", JSON.stringify(users), err => {
        // Checking for errors
        if (err) throw err;
        console.log("Done writing"); // Success
});
```

NODE JS

HTTP request using node-fetch in nodejs

A web application often needs to communicate with web servers to get various resources. In NodeJS, several packages/libraries can achieve the same result. One of them is the node-fetch package. node-fetch is a lightweight module that enables us to use the fetch() function in NodeJS, with very similar functionality as window.fetch() in native JavaScript.

To use the module in code, use:

```
const fetch = require('node-fetch’);
```

Followed by

```
fetch(url[, options]);
```

The url parameter is simply the direct URL to the resource we wish to fetch. It has to be an absolute URL or the function will throw an error.

HTTP request using node-fetch in nodejs

The function returns a Response object that contains useful functions and information about the HTTP response, such as:

- text() - returns the response body as a string
- json() - parses the response body into a JSON object, and throws an error if the body can't be parsed
- status and statusText - contain information about the HTTP status code
- ok - equals true if status is a 2xx status code (a successful request)
- headers - an object containing response headers, a specific header can be accessed using the get() function.

HTTP request using node-fetch in nodejs

PES UNIVERSITY ONLINE

Sending GET Requests Using node-fetch

```
const fetch = require('node-fetch’);
 fetch('https://google.com’)
 .then(res => res.text())
.then(text => console.log(text))
```

In the code above, we're loading the node-fetch module and then fetching the Google home page. The only parameter we've added to the fetch() function is the URL of the server we're making an HTTP request to. Because node-fetch is promise-based, we're chaining a couple of .then() functions to help us manage the response and data from our request.

HTTP request using node-fetch in nodejs

Fetching JSON Data From REST API

```
const fetch = require('node-fetch');
fetch('https://jsonplaceholder.typicode.com/users’)
.then(res => res.json())
.then(json => { console.log("First user in the array:");
 console.log(json[0]);
console.log("Name of the first user in the array:");
console.log(json[0].name); })
```

The body of the HTTP response contains JSON-formatted data, namely an array containing information about users. With this in mind, we used the .json() function, and this allowed us to easily access individual elements and their fields.

Sending POST Requests Using *node-fetch*

We can also use the fetch() function to post data instead of retrieving it. An additional parameter (method, body or headers etc)is to be added to make POST requests to a web server. Without this optional parameter, our request is a GET request, by default.

- Method - sets what type of HTTP request we're using (POST in our case),
- body - contains the body/data of our request, and
- headers -contains all the necessary headers, which in our case is just the Content-Type.

HTTP request using node-fetch in nodejs

Eg – to add a new item to JSONPlaceholder's todos list for the user whose id equals 123. First, we need to create a todo object, and later convert it to JSON when adding it to the body field:

```
const fetch = require('node-fetch');
let todo = {
    userId: 123,
    title: "loren impsum doloris",
    completed: false
};
fetch('https://jsonplaceholder.typicode.com/todos', {
    method: 'POST',
    body: JSON.stringify(todo),
    headers: { 'Content-Type': 'application/json' }
}).then(res => res.json())
  .then(json => console.log(json));
```

NODE JS

HTTP request using node-fetch in nodejs

Eg – to add a new item to JSONPlaceholder's todos list for the user whose id equals 123. First, we need to create a todo object, and later convert it to JSON when adding it to the body field:

```
const fetch = require('node-fetch');
let todo = {
    userId: 123,
    title: "loren impsum doloris",
    completed: false
};
fetch('https://jsonplaceholder.typicode.com/todos', {
    method: 'POST',
    body: JSON.stringify(todo),
    headers: { 'Content-Type': 'application/json' }
}).then(res => res.json())
  .then(json => console.log(json));
```

We called the fetch() function, with the appropriate URL and we set the necessary options using the optional parameter of the fetch() function. We used JSON.stringify() to convert our object to a JSON-formatted string before sending it to the web server. Then, same as with retrieving data - we awaited the response, converted it to JSON, and printed it to the console.

NODE JS

HTTP request using node-fetch in nodejs

```
const fetch = require('node-fetch');
function checkResponseStatus(res) {
    if(res.ok){
        return res
    } else {
        throw new Error(`The HTTP status of the reponse: ${res.status}
(${res.statusText})`);
    }
}

fetch('https://jsonplaceholder.typicode.com/MissingResource')
    .then(checkResponseStatus);
    .then(res => res.json());
    .then(json => console.log(json));
    .catch(err => console.log(err));
```

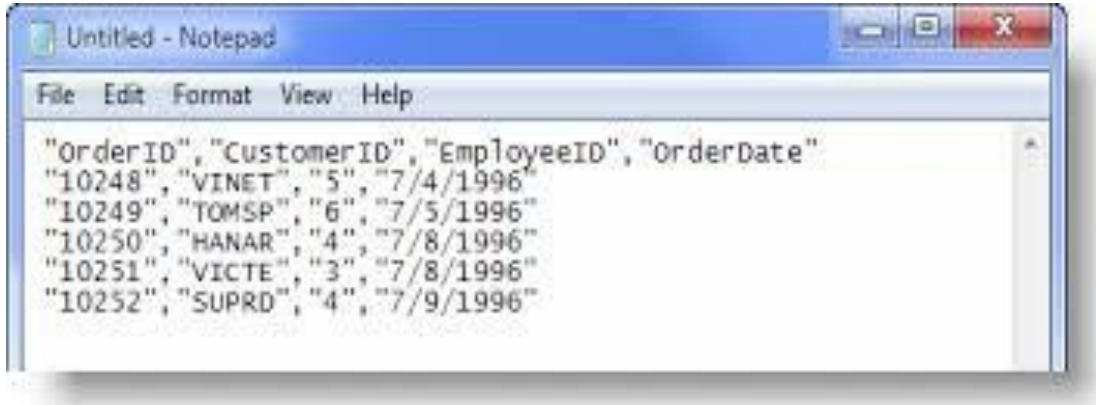
MongoDB

File Based Systems

SQL Database
• Relational database
• Supports SQL query language
• Table based
• Row based
• Column based
• Contains schema which is predefined
• Not fit for hierarchical data storage
• Vertically scalable - increasing RAM
• Emphasizes on ACID properties (Atomicity, Consistency, Isolation and Durability)

File Based Systems

File (Typically a CSV file)



Database

Emp_name	Emp_id	Emp_addr	Emp_desig	Emp_Sal
Prasad	100	"Shubhodaya", Near Katariguppe Big Bazaar, BSK II stage, Bangalore	Project Leader	40000
Usha	101	#165, 4th main Chamrajpet, Bangalore	Software engineer	10000
Nupur	102	#12, Manipal Towers, Bangalore	Lecturer	30000
Peter	103	Syndicate house, Manipal	IT executive	15000

NoSQL

- Basically a database used to manage huge sets of unstructured data, where in the data is not stored in tabular relations like relational databases.
- NoSQL plays a vital role in an enterprise application which needs to access and analyze a massive set of data that is being made available on multiple virtual servers (remote based) in the cloud infrastructure and mainly when the data set is not structured. Hence, the NoSQL database is designed to overcome the Performance, Scalability, Data Modelling and Distribution limitations that are seen in the Relational Databases.

NoSQL Database Types

- Document Databases : In this type, key is paired with a complex data structure called as Document. Example : MongoDB
- Graph stores : This type of database is usually used to store networked data. Where in we can relate data based on some existing data.
- Key-Value stores : These are the simplest NoSQL databases. In this each is stored with a key to identify it. In some Key-value databases, we can even save the type of the data saved along, like in Redis.
- Wide-column stores : Used to store large data sets(store columns of data together). Example : Cassandra(Used in Facebook), HBase etc.

MongoDB
Introduction

- Name comes from “Humongous” & huge data
- Written in C++, developed in 2009

- MongoDB is an open source, document-oriented database designed with both scalability and developer agility in mind
- Instead of storing your data in tables and rows as you would with a relational database, in MongoDB you store JSON-like documents with dynamic schemas (schema-free, schemaless)

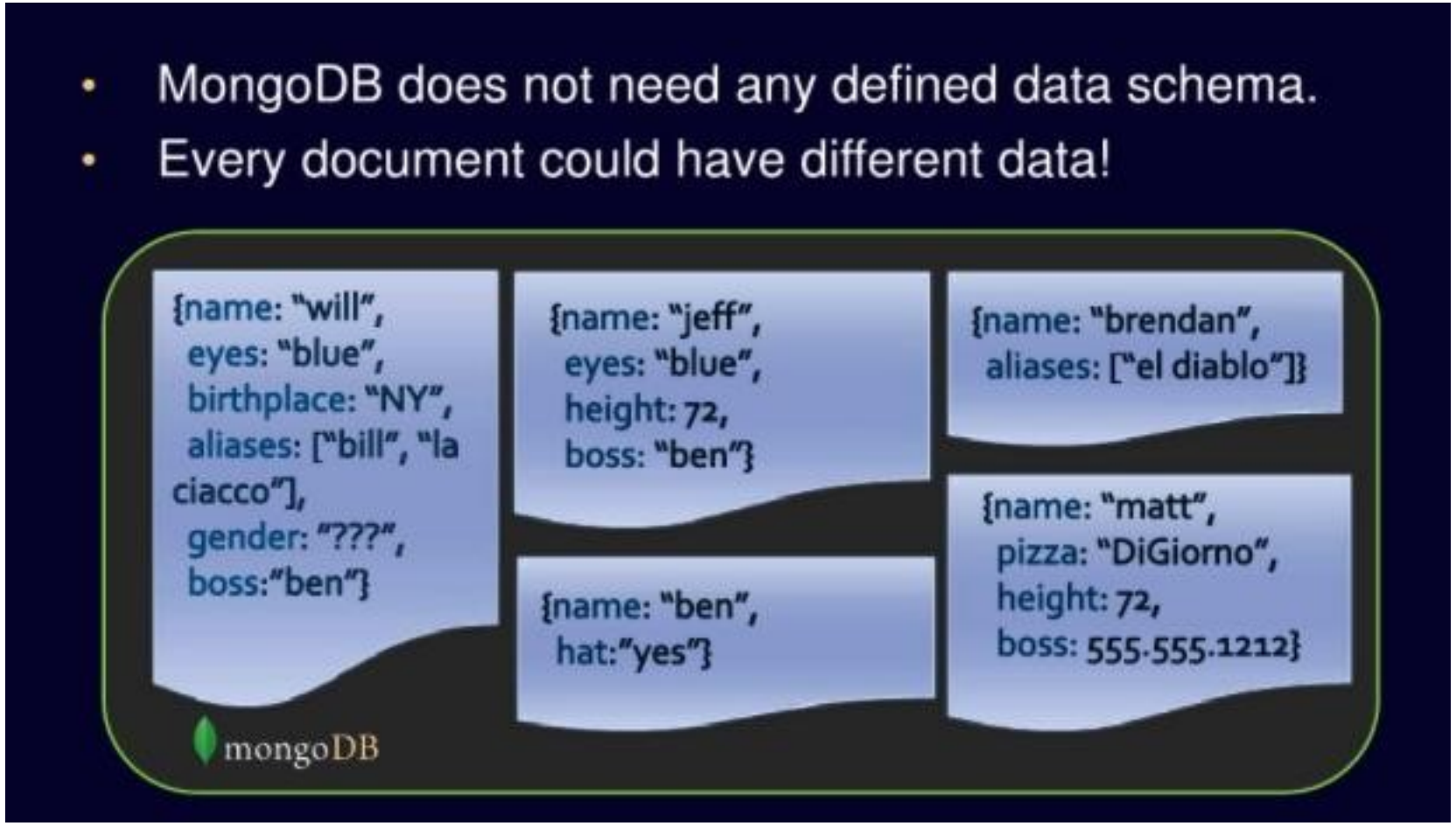
MongoDB
NoSQL Database

PES UNIVERSITY ONLINE

- Stands for **N**ot **O**nly **SQL**??
- Class of non-relational data storage systems
- Usually do not require a fixed table schema nor do they use the concept of joins to derive data from different tables

MongoDB – a NoSQL Database

- No Defined Schema (Schema Free or Schema Less)



MongoDB

Terms Mapping (DB vs. MongoDB)

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
Column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default _id key provided by mongodb)

MongoDB

Data Model – BSON Format

- BSON format (binary JSON)
- Developers can easily map to modern object-oriented languages without a complicated ORM (Object Relational mapping)layer.

```
{ author: 'joe',
  created : new Date('03/28/2009'),
  title : 'Yet another blog post',
  text : 'Here is the text...',
  tags : [ 'example', 'joe' ],
  comments : [
              { author: 'jim',
                comment: 'I disagree'
              },
              { author: 'nancy',
                comment: 'Good post'
              }
  ]
}
```

Remember it is stored in binary formats

```
"\x16\x00\x00\x00\x02hello\x00
 \x06\x00\x00\x00world\x00\x00"

"1\x00\x00\x00\x04BSON\x00&\x00
 \x00\x00\x020\x00\x08\x00\x00
 \x00awesome\x00\x011\x00333333
 \x14@\x102\x00\xc2\x07\x00\x00
 \x00\x00"
```

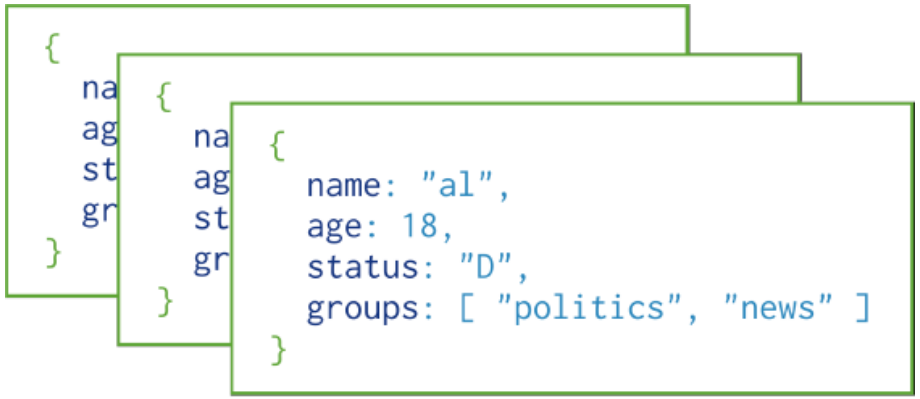
MongoDB

MongoDB Data Model

One ***document*** (e.g., one tuple in RDBMS)

```
{
  name: "sue",          <-- field: value
  age: 26,              <-- field: value
  status: "A",          <-- field: value
  groups: [ "news", "sports" ]   <-- field: value
}
```

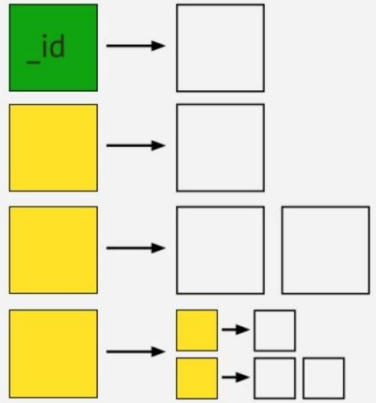
One ***Collection*** (e.g., one Table in RDBMS)



- **Collection** is a group of similar documents
- Within a collection, each document must have a unique Id

MongoDB Document

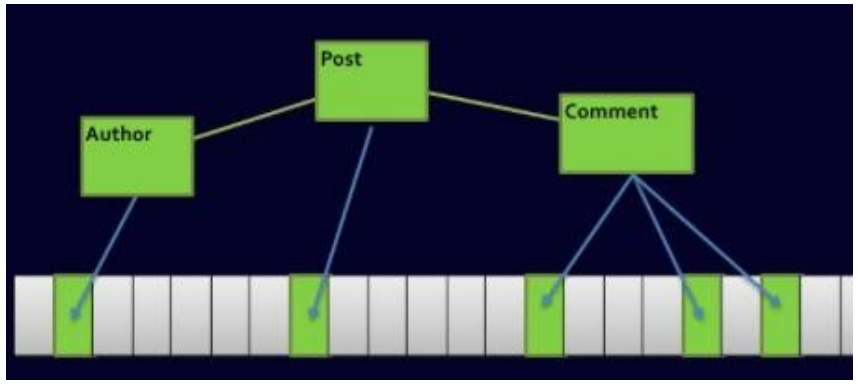
- **N-dimensional** storage
- Field can contain **many** values and **embedded** values
- Query on **any field & level**
- **Flexible** schema
- Optimal data locality requires fewer **indexes** and provides better **performance**



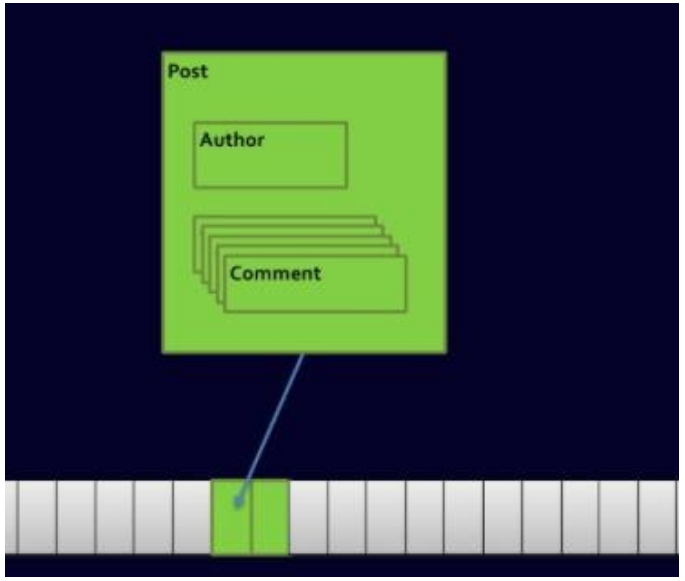
MongoDB

Complex Join Queries

Relational DBs



MongoDB



MongoDB

Getting Started…

Install it

Practice simple stuff

Move to complex stuff

Install it from here: http://www.mongodb.org

MongoDB

Data Operations

Create

db.collection.insert(<document>)

db.collection.save(<document>)

db.collection.update(<query>, <update>, { upsert: true })

Read

db.collection.find(<query>, <projection>)

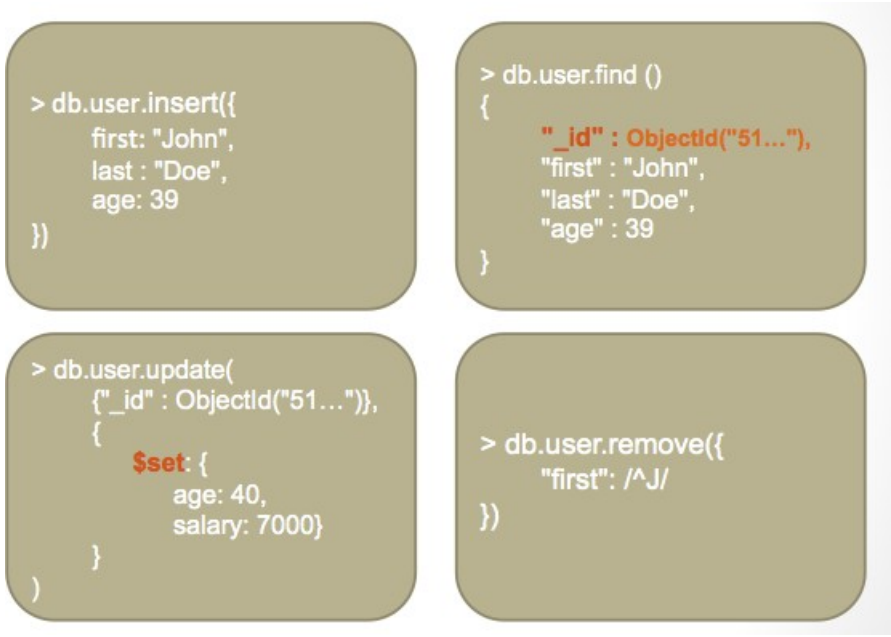
db.collection.findOne(<query>, <projection>)

Update

db.collection.update(<query>, <update>, <options>)

Delete

db.collection.remove(<query>, <justOne>)



Example Operations – Creation and Deletion

In RDBMS

```
CREATE TABLE users (
    id MEDIUMINT NOT NULL
        AUTO_INCREMENT,
    user_id Varchar(30),
    age Number,
    status char(1),
    PRIMARY KEY (id)
)
```

```
DROP TABLE users
```

In MongoDB

Either insert the 1st document

```
db.users.insert( {
    user_id: "abc123",
    age: 55,
    status: "A"
} )
```

Or create "Users" collection explicitly

```
db.createCollection("users")
```

```
db.users.drop()
```

MongoDB

Example Operations – Removal or Deletion

- You can put condition on any field in the document (even ***_id***)

```
db.users.remove(          <-- collection
   { status: "D" }        <-- remove criteria
)
```

The following diagram shows the same query in SQL:

```
DELETE FROM users         <-- table
WHERE   status = 'D'      <-- delete criteria
```

db.users.remove ( ) → Removes all documents from *users* collection

MongoDB

Example Operations – Update

```
db.users.update(                    <-- collection
   { age: { $gt: 18 } },            <-- update criteria
   { $set: { status: "A" } },       <-- update action
   { multi: true }                  <-- update option
)
```

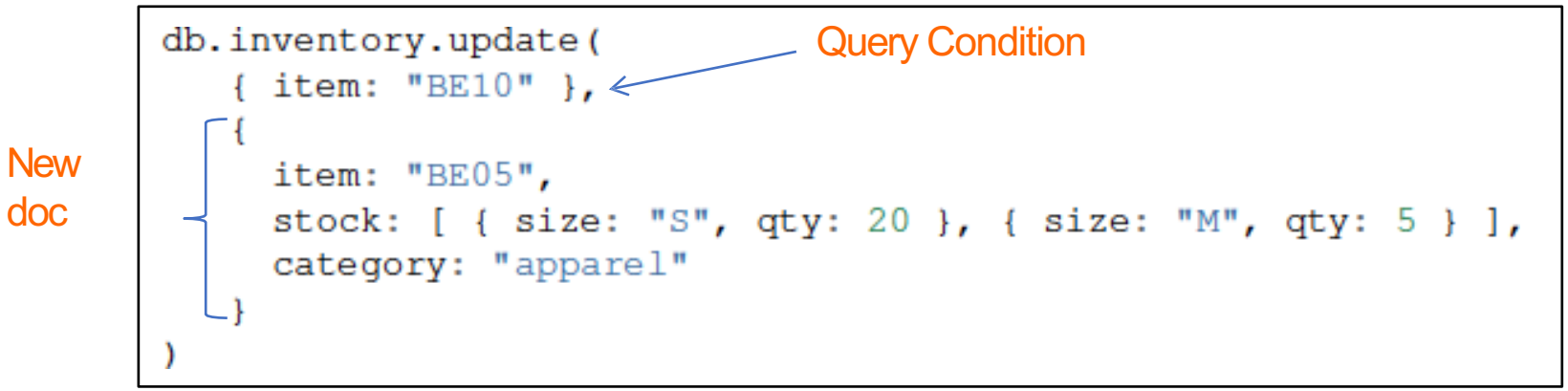
Otherwise, it will update only the 1st matching document

Equivalent to in SQL:

```
UPDATE users          <-- table
SET    status = 'A'   <-- update action
WHERE  age > 18       <-- update criteria
```

MongoDB

Example Operations – Replace a document



For the document having item = “BE10”, replace it with the given document

MongoDB

Example Operations – Insert or Update?

```
db.inventory.update(
   { item: "TBD1" },
   {
     item: "TBD1",
     details: { "model" : "14Q4", "manufacturer" : "ABC Company" },
     stock: [ { "size" : "S", "qty" : 25 } ],
     category: "houseware"
   },
   { upsert: true }
)
```

The ***upsert*** option

If the document having item = “TBD1” is in the DB, it will be replaced
Otherwise, it will beinserted.

Node JS and MongoDB Connectivity

- Node.js can use this native module to manipulate MongoDB databases

  require('mongodb');

- Alternately, use more sophisticated third party module like 'mongoose' that provides Object Data Modeling capabilities

Creating a Database

- To create a database in MongoDB, start by creating a MongoClient object, then specify a connection URL with the correct ip address and the name of the database.
- MongoDB will create the database if it does not exist, and make a connection to it.

Node JS and MongoDB Connectivity

```
var MongoClient = require('mongodb').MongoClient;

//Create a database named "mydb":

var url = "mongodb://localhost:27017/mydb";


MongoClient.connect(url, {useUnifiedTopology:true},function(err, db) {
  if (err) throw err;
  console.log("Database created!");
  db.close();
});
```

Creating a Collection

- To create a collection in MongoDB, use the Collection() method
- In MongoDB, a collection is not created until it gets content

Insert a single document Into Collection

- To insert document into a collection, use the insertOne() method
- A document in MongoDB is the same as a record in MySQL
- The first parameter of the insertOne() method is an object containing the name(s) and value(s) of each field in the document you want to insert.
- It also takes a callback function where you can work with any errors, or the result of the insertion
- To insert multiple documents at once, use insertMany() method

Node JS and MongoDB Connectivity


```
//Create collection
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.createCollection("customers", function(err, res) {
    if (err) throw err;
    console.log("Collection created!");
    db.close();
  });
});
```

Node JS and MongoDB Connectivity


```
//insert one document
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myobj = { name: "Company Inc", address: "Highway 37" };
  dbo.collection("customers").insertOne(myobj, function(err,
res) {
    if (err) throw err;
    console.log("1 document inserted");
    db.close();
  });
});
```

Node JS and MongoDB Connectivity

PES UNIVERSITY ONLINE

```
//insert many document
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myobj = [
    { name: 'John', address: 'Highway 71'},
    { name: 'Peter', address: 'Lowstreet 4'},
    { name: 'Amy', address: 'Apple st 652'},
    { name: 'Hannah', address: 'Mountain 21'}]
dbo.collection("customers").insertMany(myobj, function(err, res) {
    if (err) throw err;
    console.log("Number of documents inserted: " + res.insertedCount);
    db.close();
  });
});
```

Select the documents from collection:

- In MongoDB use the find() and findOne() methods to find data in a collection.
- Just like the SELECT statement is used to find data in a table in a MySQL database.

Find:

- To select data from a table in MongoDB, we can also use the find() method.
- The find() method returns all occurrences in the selection.
- The first parameter of the find() method is a query object.

Node JS and MongoDB Connectivity


```
//find one document
var MongoClient
=require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.collection("customers").findOne({},
function(err, result) {
    if (err) throw err;
    console.log(result.name);
    db.close();
  });
});
```

Node JS and MongoDB Connectivity


```
//find all documents
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.collection("customers").find({}).toArray(function(err,
result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

Node JS and MongoDB Connectivity


```
//find all documents with selection of fields
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.collection("customers").find({}, { projection: { _id: 0,
name: 1, address: 1 } }).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

Node JS and MongoDB Connectivity


```
//update a document
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myquery = { address: "Valley 345" };
  var newvalues = { $set: {name: "Mickey", address: "Canyon 123" } };
  dbo.collection("customers").updateOne(myquery, newvalues, function(err,
res) {
    if (err) throw err;
    console.log("1 document updated");
    db.close();
  });
});
```

Node JS and MongoDB Connectivity


```
//update specific field
var myquery = { address: "Valley 345" };
  var newvalues = { $set: { address: "Canyon 123" } };
  dbo.collection("customers").updateOne(myquery, newvalues, function(err,
res) {
...
```

Node JS and MongoDB Connectivity

```
//delete specific field
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myquery = { address: 'Mountain 21' };
  dbo.collection("customers").deleteOne(myquery, function(err, obj) {
    if (err) throw err;
    console.log("1 document deleted");
    db.close();
  });
});
```

Node JS and MongoDB Connectivity


```
//delete MANY fields
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myquery = { address: /^O/ };  // address starts with O
  dbo.collection("customers").deleteMany(myquery, function(err, obj) {
    if (err) throw err;
    console.log(obj.result.n + " document(s) deleted");
    db.close();
  });
});
```

Node JS and MongoDB Connectivity

```
//drop a collection
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.collection("customers").drop(function(err, delOK) {
    if (err) throw err;
    if (delOK) console.log("Collection deleted");
    db.close();
  });
});
```

Node JS and MongoDB Connectivity

PES UNIVERSITY ONLINE

```
//drop a collection using dropcollection

var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.dropCollection("customers", function(err, delOK) {
    if (err) throw err;
    if (delOK) console.log("Collection deleted");
    db.close();
  });
});
```

NODE JS

Buffers Introduction

- Pure JavaScript is Unicode friendly, but it is not so for binary data.
- Working with TCP streams or the file system, it's necessary to handle octet streams.
- 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.

NODE JS

Buffer Operations

- Creating Buffers
- Writing to Buffers
- Reading from Buffers
- Concatenate Buffers
- Copy Buffers
- Compare Buffers

Creating Buffers

- The **Buffer.alloc() method** is used to create a new buffer object of the specified size.
- This method is slower than **Buffer.allocUnsafe() method** but it assures that the newly created Buffer instances will never contain old information or data that is potentially sensitive.

Syntax

Buffer.alloc(size, fill, encoding)

size: It specifies the size of the buffer.
fill: It is an optional parameter and specifies the value to fill the buffer. Its default value is 0.
encoding: It is an optional parameter that specifies the value if the buffer value is a string. Its default value is **'utf8'**.
Return Value: This method returns a new initialized Buffer of the specified size.
A TypeError will be thrown if the given size is not a number.

Creating Buffers

You can create a buffer using a given array using **from()** or using an instance of a buffer;
for example,

let's initialize a buffer with the contents of the array **[10, 20, 30, 40, 50]**:

```
Var buf3 = new Buffer.alloc(5,[10, 20, 30, 40, 50]);
Var buf4 = Buffer.from([ 10, 20, 30, 40, 50]);
console.log(buf3)
console.log(buf4)
```

The integers that make up the array's contents represent bytes.

Writing to Buffers

Syntax

buf.write(string[, offset][, length][, encoding]) Parameters

string – This is the string data to be written to buffer.
offset – This is the index of the buffer to start writing at. Default value is 0.
length – This is the number of bytes to write. Defaults to buffer.length.
encoding – Encoding to use. 'utf8' is the default encoding.

Return Value

This method returns the number of octets written. If there is not enough space in the buffer to fit the entire string, it will write a part of the string.

Eg –

```
len = buf5.write("Packt student", "utf-8")
console.log (len) //The length becomes 13 after writing into the buffer
```

Reading from Buffers

Syntax

buf.toString([encoding][, start][, end]) Parameters

encoding – Encoding to use. 'utf8' is the default encoding.
start – Beginning index to start reading, defaults to 0.
end – End index to end reading, defaults is complete buffer.

Return Value

This method decodes and returns a string from buffer data encoded using the specified character set encoding.
Eg -

```
console.log(buf5.toString("utf-8",0,13))
```

NODE JS

Compare Buffers

buf.compare(target[, targetStart[, targetEnd[, sourceStart[, sourceEnd]]]])

- target <Buffer> | <Uint8Array> A Buffer or Uint8Array with which to compare buf.
- targetStart <integer> The offset within target at which to begin comparison. Default: 0.
- targetEnd <integer> The offset within target at which to end comparison (not inclusive). Default: target.length.
- sourceStart <integer> The offset within buf at which to begin comparison. Default: 0.
- sourceEnd <integer> The offset within buf at which to end comparison (not inclusive). Default: buf.length.
- Returns: <integer>

Compares buf with target and returns a number indicating whether buf comes before, after, or is the same as target in sort order.

0 is returned if target is the same as buf

1 is returned if target should come before buf when sorted.

-1 is returned if target should come after buf when sorted.

NODE JS

Compare Buffers

```
// Node.js program to demonstrate the
// Buffer.compare() Method

// Creating a buffer
var buffer1 = Buffer.from('bufferworld');
var buffer2 = Buffer.from(' bufferworld ');
var op = Buffer.compare(buffer1, buffer2);
console.log(op);

var buffer1 = Buffer.from('Java');
var buffer2 = Buffer.from('Python');
var op = Buffer.compare(buffer1, buffer2);
console.log(op);
```

NODE JS

Copy Buffers

buf.copy(target[, targetStart[, sourceStart[, sourceEnd]]])#

- target <Buffer> | <Uint8Array> A Buffer or Uint8Array to copy into.
- targetStart <integer> The offset within target at which to begin writing. Default: 0.
- sourceStart <integer> The offset within buf from which to begin copying. Default: 0.
- sourceEnd <integer> The offset within buf at which to stop copying (not inclusive).

  Default: buf.length.
- Returns: <integer> The number of bytes copied.

Copies data from a region of buf to a region in target, even if the target

memory region overlaps with buf.

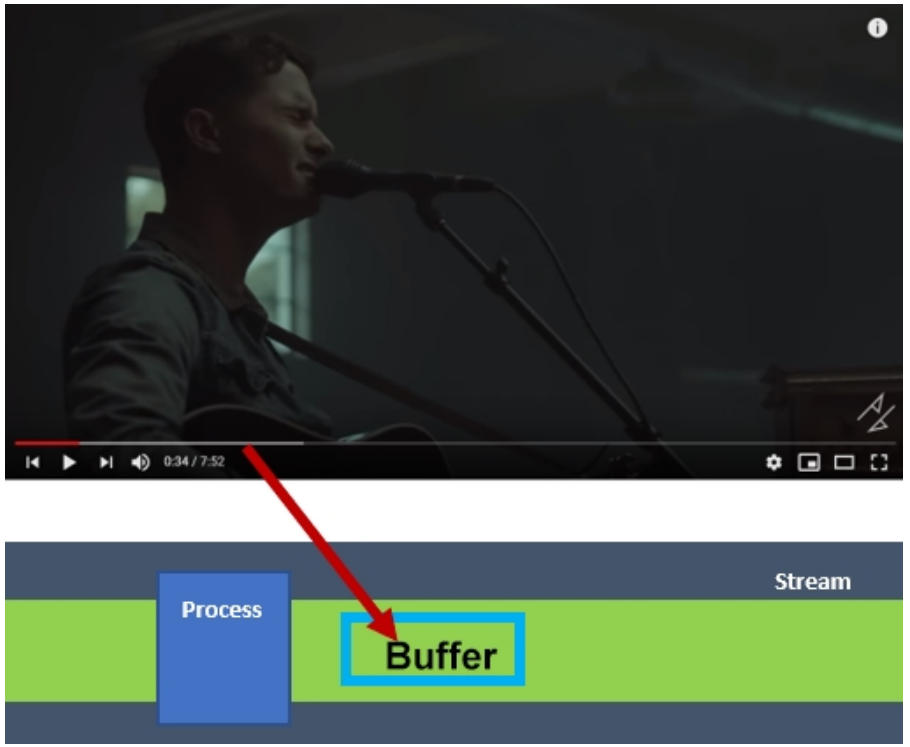
NODE JS

Copy Buffers

```
// Node.js program to demonstrate the
// Buffer.copy() Method

// Creating a buffer
var buffer2 = Buffer.from('for');

var buffer1 = Buffer.from(‘demo for buffercopy');

buffer2.copy(buffer1, 5, 0);

console.log(buffer1.toString());
```

NODE JS

Buffers and Streams in Action – YouTube Example



NODE JS

Streams

- Streams are one of the fundamental concepts that power Node.js applications.
- They are data-handling method and are used to read or write input into output sequentially.
- Streams are a way to handle reading/writing files, network communications, or any kind of end-to-end information exchange in an efficient way.
- A program reads a file into memory **all at once** like in the traditional way, whereas streams read chunks of data piece by piece, processing its content without keeping it all in memory.
- This makes streams really powerful when working with **large amounts of data**, for example, a file size can be larger than your free memory space, making it impossible to read the whole file into the memory in order to process it.
- Streams also give us the power of 'composability' in our code

For Example "streaming" services such as **YouTube or Netflix**

Why Streams

Streams basically provide two major advantages compared to other data handling methods:

Memory efficiency: you don’t need to load large amounts of data in memory before you are able to process it

Time efficiency: it takes significantly less time to start processing data as soon as you have it, rather than having to wait with processing until the entire payload has been transmitted

Types of streams in Node Js

There are 4 types of streams in Node.js:

Writable: streams to which we can write data. For example, fs.createWriteStream() lets us write data to a file using streams.

Readable: streams from which data can be read. For example: fs.createReadStream() lets us read the contents of a file.

Duplex: streams that are both Readable and Writable. For example, net.Socket

Transform: streams that can modify or transform the data as it is written and read. For example, in the instance of file-compression, you can write compressed data and read decompressed data to and from a file.

In HTTP server, **request is a readable stream and response is a writable stream.**

Streams as Events

A stream is said to be readable if it permits data to be read from a source, irrespective of what the source is, be it another stream, a file in a filesystem, a buffer in memory, and so on. Various data events can be emitted at various points in a stream. Thus, streams can also be referred to as instances of **EventEmitters**. Listening to a data event and attaching a callback are the best ways to read data from a stream. A readable stream emits a data event, and your callback executes when data is available.

NODE JS

Streams as Events

```
var fs = require("fs");
var data = '';
// Create a readable stream
var readerStream = fs.createReadStream('fileread.txt');
// Set the encoding to be utf8.
readerStream.setEncoding('UTF8');
// Handle stream events --> data, end, and error
readerStream.on('data', function(chunk) {
   data += chunk;
});
readerStream.on('end',function() {
   console.log(data);
});
readerStream.on('error', function(err) {
   console.log(err.stack);
});
console.log("Program Ended");
```

Streams as Events

A stream is said to be writeable if it permits data to be written to a destination, irrespective of what the destination is. It could be another stream, a file in a filesystem, a buffer in memory, and so on. Similar to readable streams, various events that are emitted at various points in writeable streams can also be referred to as instances of **EventEmitter**. The **write()** function is available on the **stream** instance. It makes writing data to a stream possible.

NODE JS

Streams as Events

```
var data = 'Simply Easy Learning';
// Create a writable stream
var writerStream = fs.createWriteStream('output.txt');
// Write the data to stream with encoding to be utf8
writerStream.write(data,'UTF8');
// Mark the end of file
writerStream.end();
// Handle stream events --> finish, and error
writerStream.on('finish', function() {
   console.log("Write completed.");
});
writerStream.on('error', function(err) {
   console.log(err.stack);
});

console.log("Program Ended");
```

Piping streams

Piping is a mechanism where we provide the output of one stream as the input to another stream. It is normally used to get data from one stream and to pass the output of that stream to another stream. There is no limit on piping operations.

```
var fs = require("fs");
// Create a readable stream
var readerStream = fs.createReadStream('input.txt');
// Create a writable stream
var writerStream = fs.createWriteStream('output.txt');
// Pipe the read and write operations
// read input.txt and write data to output.txt
readerStream.pipe(writerStream);
console.log("Program Ended");
```

Chaining the Streams

Chaining is a mechanism to connect the output of one stream to another stream and create a chain of multiple stream operations. It is normally used with piping operations.

```
var fs = require("fs");
var zlib = require('zlib');

// Compress the file input.txt to input.txt.gz
fs.createReadStream('input.txt')
   .pipe(zlib.createGzip())
   .pipe(fs.createWriteStream('input.txt.gz'));

console.log("File Compressed.");
```

Chaining the Streams

```
var fs = require("fs");
var zlib = require('zlib');

// Decompress the file input.txt.gz to input.txt
fs.createReadStream('input.txt.gz')
  .pipe(zlib.createGunzip())
  .pipe(fs.createWriteStream('input.txt'));

console.log("File Decompressed.");
```

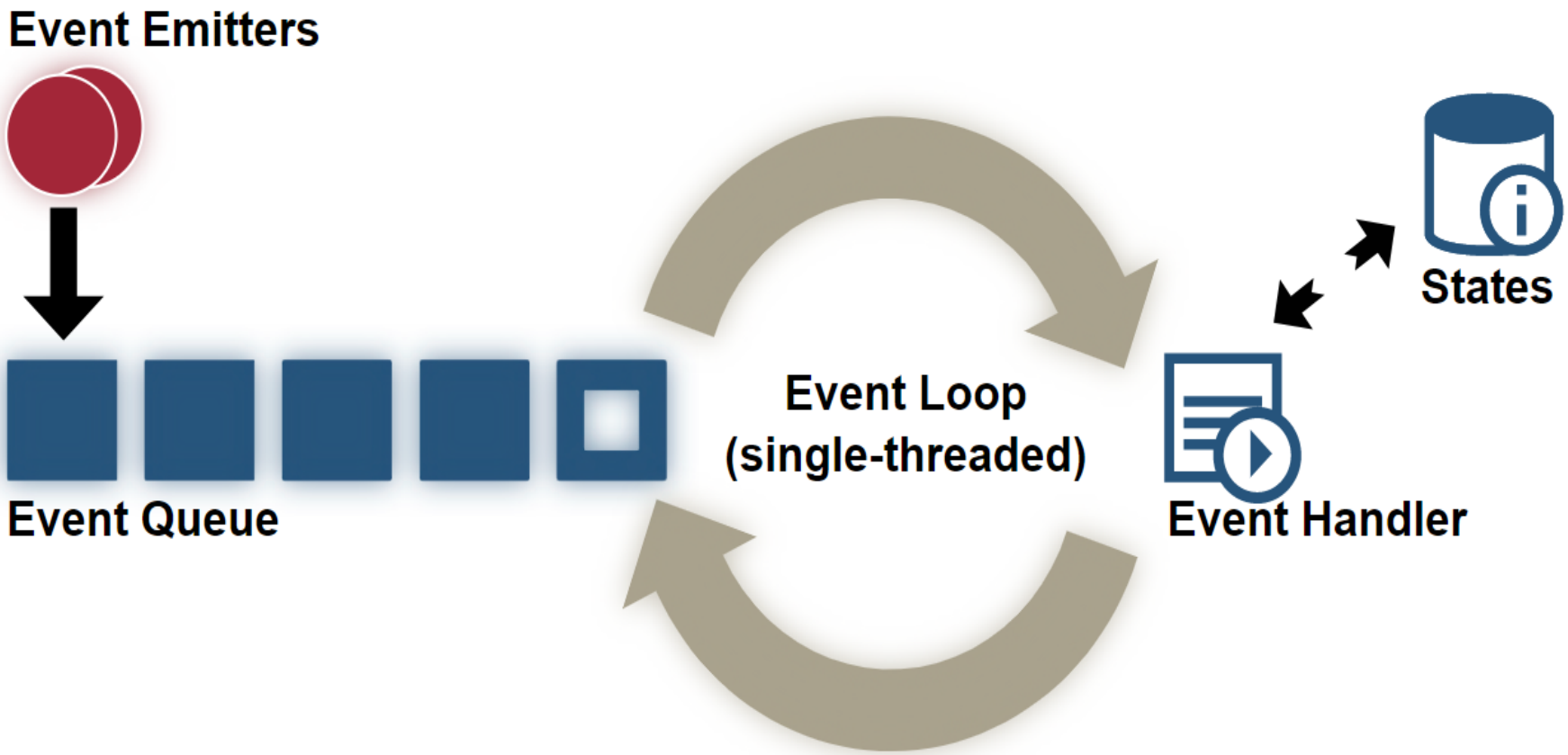
NODE JS

Native Nodejs Objects - streams

Readable Streams	Writable Streams
HTTP responses, on the client	HTTP requests, on the client
HTTP requests, on the server	HTTP responses, on the server
fs read streams	fs write streams
zlib streams	zlib streams
crypto streams	crypto streams
TCP sockets	TCP sockets
child process stdout and stderr	child process stdin
process.stdin	process.stdout, process.stderr

NODE JS

Events Architecture flow



NODE JS

Events

- 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.

NODE JS

Event Driven Programming

- Node.js uses events heavily and it is also one of the reasons why Node.js is pretty fast compared to other similar technologies.
- As soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for the event to occur.
- In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected.

NODE JS

EventEmitter Class

- The functions that listen to events act as **Observers**.
- Whenever an event gets fired, its listener function starts executing. Node.js has multiple in-built events available through events module and EventEmitter class which are used to bind events and event-listeners

```
var events = require('events');

// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();
```

EventEmitter Class

EventEmitter provides multiple properties like **on** and **emit**. **on** property is used to bind a function with the event and **emit** is used to fire an event.

Methods:

- addListener(event, listener)
- on(event, listener)
- once(event, listener)
- removeListener(event, listener)
- removeAllListeners([event])
- setMaxListeners(n)
- listeners(event)
- emit(event, [arg1], [arg2], [...])

NODE JS

EventEmitter Class

```
// get the reference of EventEmitter class of events module
var events = require('events');

//create an object of EventEmitter class by using above reference
var em = new events.EventEmitter();

//Subscribe for FirstEvent
em.on('FirstEvent', function (data) {
    console.log('First subscriber: ' + data);
});

// Raising FirstEvent
em.emit('FirstEvent', 'This is my first Node.js event emitter
example.');
```

EventEmitter Class

In the above example, we first import the 'events' module and then create an object of EventEmitter class. We then specify event handler function using on() function. The on() method requires name of the event to handle and callback function which is called when an event is raised.

The emit() function raises the specified event. First parameter is name of the event as a string and then arguments. An event can be emitted with zero or more arguments. You can specify any name for a custom event in the emit() function.

NODE JS

EventEmitter Class

```
var emitter = require('events').EventEmitter;
var em = new emitter();
//Subscribe FirstEvent
em.addListener('FirstEvent', function (data) {
    console.log('First subscriber: ' + data);
});
/Subscribe SecondEvent
em.on('SecondEvent', function (data) {
    console.log('First subscriber: ' + data);
});
/
// Raising FirstEvent
em.emit('FirstEvent', 'This is my first Node.js event emitter example.');

// Raising SecondEvent
em.emit('SecondEvent', 'This is my second Node.js event emitter example.');
```

Class Methods:

- listenerCount(emitter, event)

Events:

- newListener

event − String: the event name

listener − Function: the event handler function

This event is emitted any time a listener is added. When this event is triggered, the listener may not yet have been added to the array of listeners for the event.

- removeListener

This event is emitted any time someone removes a listener. When this event is triggered, the listener may not yet have been removed from the array of listeners for the event.

Class Methods:

- listenerCount(emitter, event)

Events:

- newListener

event − String: the event name

listener − Function: the event handler function

This event is emitted any time a listener is added. When this event is triggered, the listener may not yet have been added to the array of listeners for the event.

- removeListener

This event is emitted any time someone removes a listener. When this event is triggered, the listener may not yet have been removed from the array of listeners for the event.

React Integration

In the command prompt:

1. **npx create-react-app myapp**
2. myapp is a folder where all the react modules will be downloaded. You can opt to choose any name for the folder. (lower case)
3. It will take around 7-8 minutes for downloading.
4. **Post download:** Give **cd myapp** and get to that directory to see all the downloaded files.
5. You will find a **src** folder. Inside the folder would be two important files.
   a. **Index.js** – will be automatically executed once we start the react engine. This in turn calls the App.js file.
   b. **App.js -** We will be typing the react programs in App.js file. (Don't forget to use the statement: **import React from "react" in** the program**)**
6. **START the REACT ENGINE**: **npm start**
7. Will lead you to a browser with default URL as **localhost:3000** which will show you the output.

React Router

React is a JavaScript library for building user interfaces. We can also extend it to build multi-page applications with the help of React Router. This is a third-party library that enables routing in our React apps.

Routing is the capacity to show different pages to the user. That means the user can move between different parts of an application by entering a URL or clicking on an element.

By default, React comes without routing. And to enable it in our project, we need to add a library named react-router.

Need of React Router

React Router plays an important role to display multiple views in a single page application. Without React Router, it is not possible to display multiple views in React applications. Most of the social media websites like Facebook, Instagram uses React Router for rendering multiple views.

React Router Installation

1. **react-router-dom:** It is used for web applications design.

2. **npm install react-router-dom**

Adding React Router Components:

The main Components of React Router are:

1. **BrowserRouter:** BrowserRouter is a router implementation that uses the HTML5 history API (pushState, replaceState and the popstate event) to keep your UI in sync with the URL. It is the parent component that is used to store all of the other components.
2. **Route:** Route is the conditionally shown component that renders some UI when its path matches the current URL.
3. **Link:** Link component is used to create links to different routes and implement navigation around the application. It works like HTML anchor tag.

4.Switch: Switch component is used to render only the first route that matches the location rather than rendering all matching routes. Although there is no defying functionality of SWITCH tag in our application because none of the LINK paths are ever going to coincide. But let's say we have a route (Note that there is no EXACT in here), then all the Route tags are going to be processed which start with '/' (all Routes start with /). This is where we need SWITCH statement to process only one of the statements.

```
import {BrowserRouter as Router, Route, Link, Switch} from "react-router-dom";
```

NODE JS

React Router

App.js

```
import React from 'react';
import {BrowserRouter as Router,Route} from "react-router-dom";
import Home from './Home';
import About from './About'
import Contact from './Contact'
class App extends React.Component
{
render()
{
return(
<Router>
   <div>
     <h1>React Router Example</h1>
     <Route path="/" component={Home} />
     <Route path="/about" component={About} />
     <Route path="/contact" component={Contact} />
   </div>
 </Router>
);
}
}export default App;
```

NODE JS

React Router

Home.js

```
import React from 'react'
class App extends React.Component {
  render() {
    return (
      <div>
        <h1>Home page</h1>
      </div>
    )
  }
}
export default App
```

NODE JS

React Router

Contact.js

```
import React from 'react'
class Contact extends React.Component {
  render() {
    return <h1>Contact us</h1>
  }
}
export default Contact
```

About.js

```
import React from 'react'
class About extends React.Component {
  render() {
    return <h1>Welocome to PES UNIVERSITY</h1>
  }
}
export default About
```

THANK YOU

Aruna S
Department of
Computer Science and Engineering
arunas@pes.edu