NODE JS

Prof. Vinay Joshi and Dr. Sarasvathi V
Department of Computer Science and Engineering

Acknowledgement
The slides are created from various internet resources with valuable contributions from multiple professors.

PES UNIVERSITY

NODE JS

Module System

Department of Computer Science and Engineering

Node JS Basics

Datatypes: Node.js contains various types of data types similar to JavaScript.

Boolean
Undefined
Null
String
Number

```
var a = 35;
console.log(typeof a);

// Variable store string data type
a = “PESU";
console.log(typeof a);
```

PES UNIVERSITY

```
// Variable store Boolean data type
a = true;
console.log(typeof a);

// Variable store undefined (no value) data type
a = undefined;
console.log(typeof a);
```

NODE JS

Node JS basics

Objects & Functions: Node.js objects are same as JavaScript objects i.e. the objects are similar to variable and it contains many values which are written as name : value pairs. Name and value are separated by colon and every pair is separated by comma.

Example:

```
var company = {
   Name: “PESU",
   Address: "Noida",
   Contact: "+919876543210",
   Email: “abc@gmail.com"
};

// Display the object information
console.log("Information of variable company:", company);

// Display the type of variable
console.log("Type of variable company:", typeof company);
```

Output:

Functions:

Node.js functions are defined using function keyword then the name of the function and parameters which are passed in the function.

In Node.js, we don’t have to specify datatypes for the parameters and check the number of arguments received.

Node.js functions follow every rule which is there while writing JavaScript functions.

Example:

```
function multiply(num1, num2) {

    // It returns the multiplication
    // of num1 and num2
    return num1 * num2;
}

// Declare variable
var x = 2;
var y = 3;

// Display the answer returned by
// multiply function
console.log("Multiplication of", x, "and", y, "is", multiply(x, y));
```

NODE JS

Node JS basics-REPL

REPL (READ, EVAL, PRINT, LOOP) is a computer environment similar to Shell (Unix/Linux) and command prompt.

- Node comes with the REPL environment when it is installed.
- System interacts with the user through outputs of commands/expressions used.
- It is useful in writing and debugging the codes.

The work of REPL can be understood from its full form:

Read : It reads the inputs from users and parses it into JavaScript data structure. It is then stored to memory.
Eval : The parsed JavaScript data structure is evaluated for the results.
Print : The result is printed after the evaluation.
Loop : Loops the input command. To come out of NODE REPL, press ctrl+c twice

NODE JS

Node JS basics-REPL

PES UNIVERSITY

```
C:\Users\DELL\Desktop>node
Welcome to Node.js v16.13.0.
Type ".help" for more information.
> 1+3
4
> y=20
20
> x+y
30
>
(To exit, press Ctrl+C again or Ctrl+D or type .exit)
```

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');
```

PES UNIVERSITY

What is a Module in Node.js?

- Modules are the blocks of encapsulated code that communicates with an external application on the basis of their related functionality.
- Modules can be a single file or a collection of multiples files/folders.
- The reason programmers are heavily reliant on modules is because of their re-usability as well as the ability to break down a complex piece of code into manageable chunks

PES UNIVERSITY

There are 2 types of Modules:

Built in Modules

Local Modules

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 built-in Modules

Node.js has many built-in modules that are part of the platform and comes with Node.js installation. **These modules can be loaded into the program by using the require function.**

Syntax:

var module = require('module_name');

The require() function will return a JavaScript type depending on what the particular module returns. The following example demonstrates how to use the Node.js Http module to create a web server.

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 Global objects

PES UNIVERSITY

Global Objects are the objects that are available in all modules.

Global Objects are built-in objects that are part of the JavaScript and can be used directly in the application without importing any particular module.

1.Class: Buffer

The Buffer class is an inbuilt globally accessible class that means it can be used without importing any module. The Buffer class is used to deal with binary data. Buffer class objects are used to represent binary data as a sequence of bytes.

2. **console:** It is an inbuilt global object used to print to stdout and stderr.

3. global: It is a global namespace. Defining a variable within this namespace makes it globally accessible.

var myvar

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 Modules

```
function printHello() {
   console.log( "Hello, World!");
}

// Now call above function after 2 seconds
var timeoutObj = setTimeout(printHello, 2000);
```

NODE JS

Importing your own modules

PES UNIVERSITY

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

Node JS Local-Modules

Local in modules example:

local modules are created locally in your Node.js application. Let's create a simple calculating module that calculates various operations. Create a calc.js file that has the following code:

Filename: calc.js

```
exports.add = function (x, y) {
   return x + y;
};

exports.sub = function (x, y) {
   return x - y;
};

exports.mult = function (x, y) {
   return x * y;
};
```

Node JS Modules

Since this file provides attributes to the outer world via exports, another file can use its exported functionality using the require() function.

Filename: index.js

```
var calculator = require('./calc');

var x = 50, y = 20;

console.log("Addition of 50 and 10 is "
            + calculator.add(x, y));

console.log("Subtraction of 50 and 10 is "
            + calculator.sub(x, y));

console.log("Multiplication of 50 and 10 is "
            + calculator.mult(x, y));
```

Step to run this program: Run index.js file using the following command:

node index.js

Demo for Modules

Create Modules in Node.js

- To create a module in Node.js, use **exports** keyword tells Node.js that the function can be used outside the module.
- **Create a file that you want to export**

JS calc.js > ...

```
1   exports.add = function (a, b) {
2       return a + b;
3   };
4
5   exports.sub = function (a, b) {
6       return a - b;
7   };
8
9   exports.mult = function (a, b) {
10      return a * b;
11  };
12    exports.div = function (a, b) {
13      return a / b;
14  };
```

- **Use the 'require' keyword to import the file**

JS U4L2.js > ...

```
14
15  var a = 50, b = 20;
16
17  console.log("Addition of 50 and 20 is "
18                          + calculator.add(a, b));
19
20  console.log("Subtraction of 50 and 20 is "
21                          + calculator.sub(a, b));
22
23  console.log("Multiplication of 50 and 20 is "
24                          + calculator.mult(a, b));
25
26  console.log("Division of 50 and 20 is "
27                          + calculator.div(a, b));
```

Node JS Modules

```
// Default export
export default function add(a, b) {
    return a + b;
}
// Named export
export function sub(a, b) {
    return a - b;
}
```

export default:

- You are exporting the add function as the default export. This means that when the module is imported, this function can be imported without specifying its exact name.

export:

- You are also exporting the sub function as a **named export**. This requires that it be imported by its exact name in other modules.

- import addition from './yourModule.js'; // Default import
- import { sub } from './yourModule.js'; // Named import

Node JS Built-in Modules

- Node.js has many built-in modules that are part of the platform and comes with Node.js installation.
- These modules can be loaded into the program by using the require function.
- Syntax:
- **var module = require('module_name');**
- The require() function will return a JavaScript type depending on what the particular module returns.

Node JS Modules

```
console.clear()
const assert = require('assert');

let x = 4;
let y = 5;

try {

    // Checking condition
    assert(x == y);
}
catch {

    // Error output
    console.log(
        `${x} is not equal to ${y}`);
}
```

The assert module provides a set of assertion functions for verifying invariants. If the condition is true it will output nothing else an assertion error is given by the console.

Node JS Modules

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write(req.url);
  res.end();
}).listen(8080);
```

This object has a property called "url" which holds the part of the url that comes after the domain name:

http://localhost:8080/summer

Given a URL, http://localhost:8080/pes.htm?city=Mangalore&year=2022
Write a code snippet to parse the URL and store the hostname, pathname, and the request parameters to a file named “requestlog.txt”.

Answer:

```
var myurl = url.parse(request.url)
var pathname = myurl.pathname;
var query = request.query;
var host = myurl.host;
fs.writeFile(“requestlog.txt”, host + pathname + query, function(err){
});
```

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

PES UNIVERSITY

```
{
    "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"
  }
```

Demo for Modules

- **Installation of validator module:**
- You can install this package by using this command.
  - *npm install validator*
- After installing validator module you can check your validator version in command prompt using the command.
  - *npm version validator*

NODE JS

Demo for Modules

PES UNIVERSITY

```
const validator = require('validator')

// Check whether given email is valid or not
var email = 'testmail@gmail.com'
console.log(validator.isEmail(email)) // true
email = 'testmail@'
console.log(validator.isEmail(email)) // false

// Check whether string is in lowercase or not
var name = 'john'
console.log(validator.isLowercase(name)) // true
name = 'JOHN'
console.log(validator.isLowercase(name)) // false

// Check whether string is empty or not
var name = '    '
console.log(validator.isEmpty(name)) // true
name = 'Smith'
console.log(validator.isEmpty(name)) // false

// Other functions also available in  isBoolean(), isCurrency(), isDecimal(), isJSON(),
isJWT(), isFloat(), isCreditCard(), etc.
```

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

THANK YOU

Prof. Vinay Joshi and Dr. Sarasvathi V

Department of
Computer Science and Engineering

NODE JS

Prof. Vinay Joshi and Dr. Sarasvathi V

Department of Computer Science and Engineering

Acknowledgement

The slides are created from various internet resources with valuable contributions from multiple professors.

PES UNIVERSITY

NODE JS

Buffers and Streams

Department of Computer Science and Engineering

Buffers and Streams in Action – YouTube Example

- Pure JavaScript is great with Unicode encoded strings, but it does not handle binary data very well.

- It is not problematic when we perform an operation on data at browser level but at the time of dealing with TCP stream and performing a read-write operation on the file system is required to deal with pure binary data.

- To satisfy this need Node.js use Buffer, So, we are going to know about buffer in Node.js.

Buffers and Streams in Action – YouTube Example

- **Buffers in Node.js:** The Buffer class in Node.js is used to perform operations on raw binary data.
- Generally, Buffer refers to the particular memory location in memory. Buffer and array have some similarities, but the difference is array can be any type, and it can be resizable.
- Buffers only deal with binary data, and it can not be resizable.

- Each integer in a buffer represents a byte. console.log() function is used to print the Buffer instance.

NODE JS

Buffer Operations

PES UNIVERSITY

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

Buffers and Streams in Action – YouTube Example

Methods to perform the operations on Buffer:

1 Buffer.alloc(size): It creates a buffer and allocates size to it.
2 Buffer.from(initialization) :It initializes the buffer with given data.
3 Buffer.write(data): It writes the data on the buffer.
4 toString() :It read data from the buffer and returned it.
5 Buffer.isBuffer(object): It checks whether the object is a buffer or not.
6 Buffer.length It returns the length of the buffer.

Buffers and Streams in Action – YouTube Example

How to create a buffer

- A buffer is created using the Buffer.from(), Buffer.alloc(), and Buffer.allocUnsafe() methods.
- While both alloc and allocUnsafe allocate a Buffer of the specified size in bytes, the Buffer created by alloc will be initialized with zeroes and the one created by allocUnsafe will be uninitialized.
- This means that while allocUnsafe would be quite fast in comparison to alloc, the allocated segment of memory may contain old data which could potentially be sensitive.

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.

Buffers and Streams in Action – YouTube Example


```
// Different Method to create Buffer
var buffer1 = Buffer.alloc(100);
var buffer2 = new Buffer('GFG');
var buffer3 = Buffer.from([1, 2, 3, 4]);
```

Buffers and Streams in Action – YouTube Example

Using a buffer
Access the content of a buffer

A buffer, being an array of bytes, can be accessed like an array:

JS

```
const buf = Buffer.from('Hey!')
console.log(buf[0]) //72
console.log(buf[1]) //101
console.log(buf[2]) //121
```

Buffers and Streams in Action – YouTube Example

Creating Buffers: Followings are the different ways to create buffers in Node.js:

- **Create an uninitiated buffer:** It creates the uninitiated buffer of given size.
  Syntax:

var ubuf = Buffer.alloc(5);

- The above syntax is used to create an uninitiated buffer of 5 octets.
- **Create a buffer from array:** It creates the buffer from given array.
  Syntax:

var abuf = new Buffer([16, 32, 48, 64]);

- The above syntax is used to create a buffer from given array.
- **Create a buffer from string:** It creates buffer from given string with optional encoding.
  Syntax:

var sbuf = new Buffer("Welcome", "ascii");

- The above syntax is used to create a Buffer from a string and encoding type can also be specified optionally.

Writing to Buffers

PES UNIVERSITY

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.

Buffers and Streams in Action – YouTube Example


```
// Writing data to Buffer
buffer1.write("Happy Learning");
```

Writing to Buffers in Node.js: The **buf.write() method** is used to write data into a node buffer.

Syntax:

buf.write(string, offset, length, encoding)

```
cbuf = new Buffer(256);
bufferlen = cbuf.write("Learn Programming!!!");
console.log("No. of Octets in which string is written : "+  bufferlen);
```

Buffers and Streams in Action – YouTube Example

Program to create a buffer,read data from buffer and write data into buffer

```
var buffer1 = Buffer.alloc(100);
var buffer2 = new Buffer('GFG');
var buffer3 = Buffer.from([1, 2, 3, 4]);

// Writing data to Buffer
buffer1.write("Happy Learning");

// Reading data from Buffer
var a = buffer1.toString('utf-8');
console.log(a);

// Check object is buffer or not
console.log(Buffer.isBuffer(buffer1));

// Check length of Buffer
console.log(buffer1.length);
```

Buffers and Streams in Action – YouTube Example


```
var bufferSrc = new Buffer('ABC');
var bufferDest = Buffer.alloc(3);
bufferSrc.copy(bufferDest);

var Data = bufferDest.toString('utf-8');
console.log(Data);

// Slicing data
var bufferOld = new Buffer('GeeksForGeeks');
var bufferNew = bufferOld.slice(0, 4);
console.log(bufferNew.toString());

// concatenate two buffer
var bufferOne = new Buffer('Happy Learning ');
var bufferTwo = new Buffer('With GFG');
var bufferThree = Buffer.concat([bufferOne, bufferTwo]);
console.log(bufferThree.toString());
```

Buffers and Streams in Action – YouTube Example


```
buf = new Buffer(26);
for (var i = 0 ; i < 26 ; i++) {
  buf[i] = i + 97;
}

console.log( buf.toString('ascii'));       // outputs: abcdefghijklmnopqrstuvwxyz
console.log( buf.toString('ascii',0,5));   // outputs: abcde
console.log( buf.toString('utf8',0,5));    // outputs: abcde
console.log( buf.toString(undefined,0,5)); // encoding defaults to 'utf8',
outputs abcde
```

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.

Buffers and Streams in Action – YouTube Example

You can print the full content of the buffer using the toString() method:

```
console.log(buf.toString())
```

Get the length of a buffer

Use the length property:

```
const buf = Buffer.from('Hey!')
console.log(buf.length)
```

Buffers and Streams in Action – YouTube Example

Iterate over the contents of a buffer

```
const buf = Buffer.from('Hey!')
for (const item of buf) {
  console.log(item) //72 101 121 33
}
```

Just like you can access a buffer with an array syntax, you can also set the contents of the buffer in the same way:

```
const buf = Buffer.from('Hey!')
buf[1] = 111 //o in UTF-8
console.log(buf.toString()) //Hoy!
```

NODE JS

Compare Buffers

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

- target <Buffer> | <Uint8Array> A Buffer or Uint8 Array 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.

Copy Buffers

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

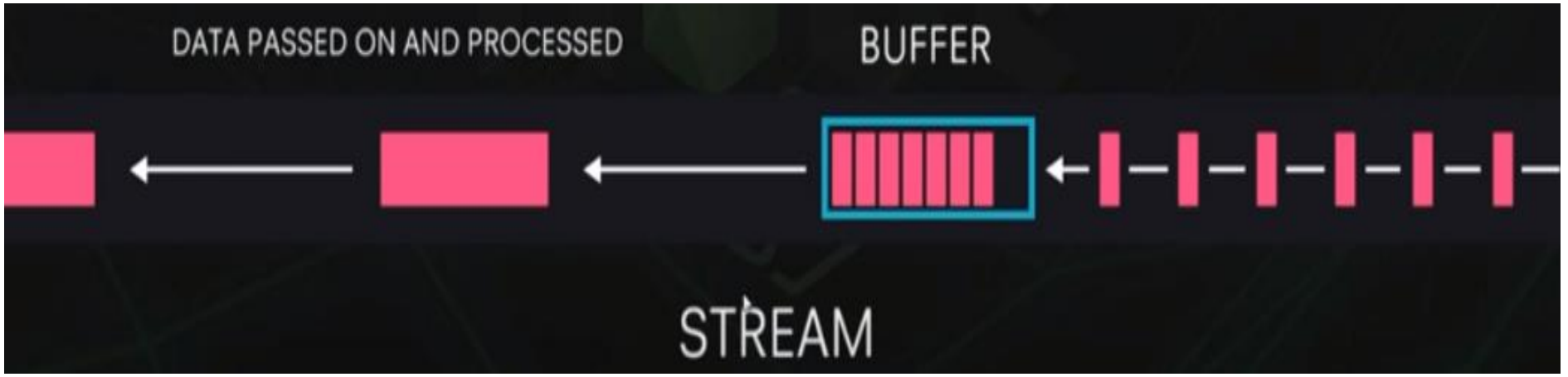
- target <Buffer> | <Uint8Array> A Buffer or Uint8Array to copy into.
- targetStart <integer> The offset within target at which to begin writing. Default: 0.
- sourceStart <integer> The offset within buf from which to begin copying. Default: 0.
- sourceEnd <integer> The offset within buf at which to 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.

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

Each type of Stream is an **EventEmitter** instance and throws several events at different instance of times.

For example, some of the commonly used events are

- **data** – This event is fired when there is data is available to read.
- **end** – This event is fired when there is no more data to read.
- **error** – This event is fired when there is any error receiving or writing data.
- **finish** – This event is fired when all the data has been flushed to underlying system.

NODE JS

Native Nodejs Objects - streams

PES UNIVERSITY

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

Buffers and Streams in Action – YouTube Example


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

const server = http.createServer(function(req, res) {
  fs.readFile(__dirname + '/data.txt', (err, data) => {
    res.end(data)
  })
})
server.listen(3000)
```

readFile() reads the full contents of the file, and invokes the callback function when it's done.

res.end(data) in the callback will return the file contents to the HTTP client.

Buffers and Streams in Action – YouTube Example

if the file is big, the operation will take quite a bit of time. Here is the same thing written using streams:

JS

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

const server = http.createServer((req, res) => {
  const stream = fs.createReadStream(__dirname + '/data.txt')
  stream.pipe(res)
})
server.listen(3000)
```

Instead of waiting until the file is fully read, we start streaming it to the HTTP client as soon as we have a chunk of data ready to be sent.

PES UNIVERSITY

pipe()

The above example uses the line stream.pipe(res): the pipe() method is called on the file stream.

What does this code do? It takes the source, and pipes it into a destination.

You call it on the source stream, so in this case, the file stream is piped to the HTTP response.

The return value of the pipe() method is the destination stream, which is a very convenient thing that lets us chain multiple pipe()

THANK YOU

Vinay Joshi and Dr.Sarasvathi V
Department of Computer Science and Engineering

NODE JS

Prof.Vinay Joshi and Dr.Sarasvathi V
Department of Computer Science and Engineering

Acknowledgement
The slides are created from various internet resources with valuable contributions from multiple professors

PES UNIVERSITY

NODE JS

File System Module

Department of Computer Science and Engineering

NODE JS

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

PES UNIVERSITY

Common use for the File System module:

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

NODE JS

Fs File Open

What is Synchronous and Asynchronous approach?

Synchronous approach: They are called blocking functions as it waits for each operation to complete, only after that, it executes the next operation, hence blocking the next command from execution i.e. a command will not be executed until & unless the query has finished executing to get all the result from previous commands.

Asynchronous approach:

- They are called non-blocking functions as it never waits for each operation to complete, rather it executes all operations in the first go itself.
- The result of each operation will be handled once the result is available i.e. each command will be executed soon after the execution of the previous command.
- While the previous command runs in the background and loads the result once it is finished processing the data.

NODE JS

Fs File Open

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

// Asynchronous read
fs.readFile('input.txt', function (err, data) {
   if (err) {
      return console.error(err);
   }
   console.log("Asynchronous read: " + data.toString());
});
```

NODE JS

Fs File Open

PES UNIVERSITY

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

// Synchronous read
var data = fs.readFileSync('input.txt');
console.log("Synchronous read: " + data.toString());
```

PES UNIVERSITY

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+,rs,rs+,w,wx,w+,wx+,a,ax,a+,ax+

Fs File Open

The fs.open() method is used to create, read, or write a file. The fs.readFile() method is only for reading the file and fs.writeFile() method is only for writing to the file, whereas fs.open() method does several operations on a file. First, we need to load the fs class which is a module to access the physical file system.

Syntax:

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

NODE JS

Fs File Open

PES UNIVERSITY

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

// Asynchronous - Opening File
console.log("opening file!");
fs.open('input.txt', 'r+', function(err, fd) {
   if (err) {
      return console.error(err);
   }
   console.log("File open successfully");
});
```

Fs File Write

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.

NODE JS

Fs File Open

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

fs.writeFile('mynewfile3.txt', 'Hello content!', function (err) {
  if (err) throw err;
  console.log('Saved!');
});
```

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

fs.open('mynewfile2.txt', 'w', function (err, file) {
  if (err) throw err;
  console.log('Saved!');
});
```

NODE JS

Fs File Open

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

console.log("writing into existing file");
fs.writeFile('input.txt', 'Web tech', function(err) {          //write data to a file
   if (err) {
      return console.error(err);
   }

   console.log("Data written successfully!");
   console.log("Let's read newly written data");

   fs.readFile('input.txt', function (err, data) {          //Read data to a file
      if (err) {
         return console.error(err);
      }
      console.log("Asynchronous read: " + data.toString());
   });
});
```

NODE JS

Fs File Open

PES UNIVERSITY

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

fs.appendFile('mynewfile1.txt', 'Hello content!', function (err) {
  if (err) throw err;
  console.log('Saved!');
});
```

Fs File Read

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).

PES UNIVERSITY

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

Fs File delete

PES UNIVERSITY

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

fs.unlink('mynewfile2.txt', function (err) {
  if (err) throw err;
  console.log('File deleted!');
});
```

THANK YOU

Prof.Vinay Joshi and Dr.Sarasvathi V

Department of Computer Science and Engineering

WEB TECHNOLOGIES

React JS – Complex Components

Prof. Vinay Joshi and Dr. Sarasvathi V
Department of Computer Science and Engineering

Acknowledgement
The slides are created from various internet resources with valuable contributions from multiple professors and teaching assistants in the university.

React.JS – Complex Components

Agenda

PES UNIVERSITY

- Introduction
- Approach followed
- Complex Component creation
- Sub component creation
- Properties from parent to sub component
- Modified parent and sub components
- Transferring properties

React.JS – Complex Components

Introduction

PES UNIVERSITY

- Combining components to create the complex one
- Advantage is **composability**
- Approach
  - Identify the major visual elements
  - Breaking them into individual components

React.JS – Components and Properties

Transferring Properties

- Sending properties down the structural tree while working with multiple components in a hierarchy

- **transferPropsTo():**
  - Passes all the properties of a parent component to a child component unless we explicitly set the value of a child component's property

- Coding examples

React.JS – Components and Properties

Transferring Properties

In modern React (including React 17 and 18), the **transferPropsTo** method is outdated and has been replaced by a more standardized approach to passing props through components. **Instead of transferPropsTo, you should use explicit prop passing.**

```
//Parent Component
import React from 'react';
import ChildComponent from './ChildComponent'; // Assume ChildComponent is in the same directory

class ParentComponent extends React.Component {
  render() {
    return (
      <div>
        <ChildComponent name="React" age={8} />
      </div>
    );
  }
}

export default ParentComponent;
```

React.JS – Components and Properties

Transferring Properties

```
//Child Component
import React from 'react';

class ChildComponent extends React.Component {
  render() {
    return (
      <div>
        <h1>Name: {this.props.name}</h1>
        <h2>Age: {this.props.age}</h2>
      </div>
    );
  }
}

export default ChildComponent;
```

React.JS – Complex Components

Approach followed

tiger | Facts, Information, & Habitat ...
britannica.com

- The complex component consists of **Image rendering, caption rendering and link rendering** on the web page

React.JS – Complex Components

Creation of complex component

```
class SrchResult extends React.Component
{
    render() {
        return (
            <div>
                <ResImage/>
                <ResCaption/>
                <ResLink/>
            </div>
        );
    }
}
```

class component names should be capitalized, and methods like render() should be correctly defined

React 18 still supports class components, though functional components with hooks are more commonly used in recent versions.

React.JS – Complex Components

Creation of sub components

```
class ResImage extends React.Component
{
  render() {
    return (
      <div>
        <img src="tiger.jpg" alt="Tiger" />
      </div>
    );
  }
}
```

In JSX, the <img> tag should be self-closing (<img />) because it doesn't have any child elements.

```
class ResLink extends React.Component {
      render() {
            return (
                  <div> <a
                  href="https://www.britannica.com/animal/tiger">
                  britannica.com </a> </div> );
      }
}
```

```
class ResCaption extends React.Component {
      render() {
            return (
                  <div> <p>tiger | Facts, Information and
                  Habitat…</p> </div> );
      }
}
```

React.JS – Complex Components

Properties from parent to sub component

To avoid hard coding, properties have to be passed from parent Components to its sub components

- src
- href
- linktext
- caption

```
<SrchResult
  src="tiger.jpg"
  href="https://www.britannica.com/animal/tiger"
  linktext="britannica.com"
  caption="tiger | Facts, Information and Habitat..."
/>
```

React.JS – Complex Components

Modified Parent and sub components

```
class SrchResult extends React.Component {
  render() {
    return (
      <div>
        <ResImage src={this.props.src} />
        <ResCaption caption={this.props.caption} />
        <ResLink                    href={this.props.href}
linktext={this.props.linktext} />
      </div>
    );
  }
}
```

```
class ResImage extends React.Component {
  render() {
    return (
      <div>
        <img src={this.props.src} alt="Tiger" />
      </div>
    );
  }
}
```

React.JS – Complex Components

Modified Parent and sub components

```
class ResCaption extends React.Component {
  render() {
    return (
      <div>
        <p>{this.props.caption}</p>
      </div>
    );
  }
}
```

```
class ResLink extends React.Component {
  render() {
    return (
      <div>
        <a href={this.props.href}>{this.props.linktext}</a>
      </div>
    );
  }
}
```

We are passing src, href, linktext and caption as props from the parent (SrchResult) to the sub-components (ResImage, ResCaption, ResLink).

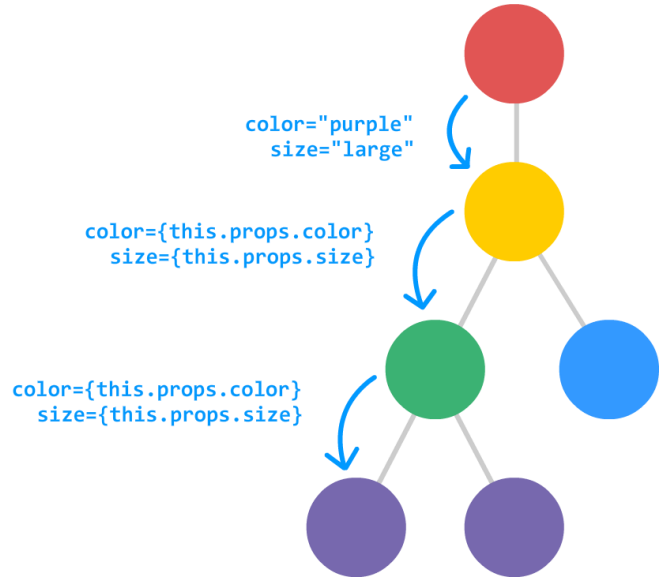
Inside class components, use **this.props** to access the properties passed from the parent.

React.JS – Complex Components

Transferring Properties

- Transferring properties was not a tedious as there was only one level of hierarchy
- If there are multiple levels of components, transferring properties will be very cumbersome
- To overcome that, we use the **spread operator (…)**

<Display {...this.props}/>



React.JS – Complex Components

Transferring Properties - Modified Parent and sub components

```
class SrchResult extends React.Component {
  render() {
    return (
      <div>
        <ResImage {...this.props} />
        <ResCaption {...this.props} />
        <ResLink {...this.props} />
      </div>
    );
  }
}
```

```
class ResImage extends React.Component {
  render() {
    return (
      <div>
        <img src={this.props.src} alt="Tiger" />
      </div>
    );
  }
}
```

Spread Operator: The {...this.props} pattern passes all props from SrchResult to its child components (ResImage, ResCaption, ResLink), avoiding the need to manually pass each prop.

Simplified Prop Transfer: This reduces redundancy and complexity, especially when there are many props or multiple levels of components.

React.JS – Complex Components

Transferring Properties - Modified Parent and sub components

```
class ResCaption extends React.Component {
  render() {
    return (
      <div>
        <p>{this.props.caption}</p>
      </div>
    );
  }
}
```

```
class ResLink extends React.Component {
  render() {
    return (
      <div>
        <a href={this.props.href}>{this.props.linktext}</a>
      </div>
    );
  }
}
```

We are passing src, href, linktext and caption as props from the parent (SrchResult) to the sub-components (ResImage, ResCaption, ResLink).

Inside class components, use **this.props** to access the properties passed from the parent.

To streamline passing props down through multiple levels of components, the spread operator (...) can be used to automatically pass all the props from the parent to the child components.

THANK YOU

Vinay Joshi and Dr.Sarasvathi V

Department of Computer Science and Engineering

vinayj@pes.edu

sarsvathiv@pes.edu

Acknowledgement

The slides are created from various internet resources with valuable contributions from multiple professors and teaching assistants in the university.

WEB TECHNOLOGIES

React JS – Component States

Prof. Vinay Joshi and Dr. Sarasvathi V
Department of Computer Science and Engineering

Acknowledgement
The slides are created from various internet resources with valuable contributions from multiple professors and teaching assistants in the university.

Component States

Agenda

PES UNIVERSITY

- Introduction
- Conventions of using states
- Demo of differences between props and states

React.JS – Stateful Components

Introduction

PES UNIVERSITY

- Components need to change based on
  - User actions (clicks, keyboard inputs, etc.)
  - Other triggers (responses received from server, timers, etc.)
- Static or stateless Components don’t undergo state changes.

React.JS – Stateful Components

Introduction

- In ReactJS, stateful components are components that maintain and manage internal state.
- Unlike stateless components, which simply render data based on props, stateful components actively respond to changes in their internal state, leading to dynamic updates of the user interface.

The state within a component can evolve based on:

1. **User actions:** Inputs such as button clicks, form entries, or keyboard interactions can trigger state changes. These updates allow the UI to react in real time, such as enabling buttons, showing validation errors, or updating content.
2. **External triggers:** Events such as receiving data from an API, completing a timer, or listening to changes in other components or systems can also cause a component's state to update. This makes the UI reactive to asynchronous events, such as displaying loading indicators or updated data upon successful API calls.

React.JS – Stateful Components

Introduction

- In contrast, **static or stateless components** are purely presentational and do not manage or track any state internally. They simply render based on the props passed down to them and do not undergo any state changes during their lifecycle.

- Stateful components offer more flexibility in managing complex logic and UI interactions, making them essential for creating dynamic, interactive applications.

- React provides hooks such as useState and useEffect to manage and handle state changes efficiently.

Component States

States - Introduction

Props	State
Immutable. once set the props cannot be changed	observable object that is to be used to hold data that may change over time and to control the behavior after each change.
Props are set by the parent component.	State is generally updated by event handlers.
Props don't have this limitation.	Used in Class Components

Both *props* and *state* are plain JavaScript objects.

- While both of them hold information that influences the output of render, they are different in their functionality with respect to component.
- Props get passed to the component similar to function parameters whereas state is managed within the component similar to variables declared within a function.

Component States

States - Introduction

- State is used with React Component Classes to make them dynamic.
- It enables the component to keep track of changing information in between renders.
- Used to track component's internal state which may change over time
- An instance of React Component Class can be defined as an object of a set of observable properties that control the behavior of the component
- An object that holds some information that may change over the lifetime of the component and to control the behavior after each change
- Can only be used in class components.
- Can only be accessed and modified inside the component and directly by the component

Component States

Conventions of using states

- Must first have some **initial state**, should define the state in the constructor of the component's class.

```
class MyClass extends React.Component {
        constructor(props)  {
            super(props);
        this.state = { attribute : "value" };    }
}
```

Component States

Conventions of using states

PES UNIVERSITY

Constructor:

- The constructor is the first method called when a class component is instantiated. It's where we initialize the component's state and bind any event handler methods if necessary.

State:

- The state object is initialized in the constructor. The state holds data that can change over time and influence how the component renders.
- State is initialized in the constructor using **this.state = {}**.

Component States

Conventions of using states

Super

Will initiate parent's constructor method and allows component to inherit methods from its parent. Call constructor of its parent class, needed to access some variables of its parent class.

- The **super(props)** call invokes the constructor of the parent class (**React.Component**). This is necessary because React components that are class-based need to call the parent's constructor to properly initialize and access **this.props**.

- Without calling **super(props)**, **this.props** will be undefined within the constructor.

Component States

Conventions of using states

PES UNIVERSITY

- **The state object can contain as many properties as you like**
- State should **never be updated explicitly**. Use **setstate()**
  - Takes a single parameter and expects an object which should contain the set of values to be updated.
  - Once the update is done the method implicitly calls the render() method to repaint the page.

setState() *can take either an object or a function.* ***setState()*** *merges the new state with the existing one and schedules a re-render of the component to reflect the changes in the UI.*

Component States

Demo of differences : props and state

- Generate the Digital clock using reactJS

THANK YOU

Vinay Joshi and Dr.Sarasvathi V

Department of Computer Science and Engineering

vinayj@pes.edu

sarsvathiv@pes.edu

Acknowledgement

The slides are created from various internet resources with valuable contributions from multiple professors and teaching assistants in the university.

WEB TECHNOLOGIES

React JS – Component States and LifeCycle Methods

Prof. Vinay Joshi and Dr. Sarasvathi V
Department of Computer Science and Engineering

Acknowledgement
The slides are created from various internet resources with valuable contributions from multiple professors and teaching assistants in the university.

Component States and LifeCycle Methods

Agenda

- Introduction to Life cycle
- Pictorial Representation of methods
- Life cycle methods in brief
- Key Methods in the React Component Lifecycle: From Mounting to Unmounting
- Demo

React.JS – Stateful Components

Counter Component

- Let’s consider this Component that shows the number of seconds the user has been on the page

122

seconds since you loaded the page

React.JS – Stateful Components

Counter Component…(cntd.)

- We consider two Components, CounterDisplay and Counter

```
ReactDOM.render(
        <CounterDisplay/>,
        document.querySelector("#container")
);
```

```
class CounterDisplay extends React.Component {
        render() {
                return (
                        <div>
                                <Counter/>
                                <h2>seconds </h2>
                                <h2>since you loaded the page</h2>
                        </div>
                );
        }
}
```

```
class Counter extends React.Component {
        render() {
                return (
                        …
                );
        }
}
```

React.JS – Stateful Components

Counter Component…(cntd.)

- We use the ***constructor*** method to initialize the counter
- The Component also has the method ***componentDidMount*** that can be used to start the counter
- It also exposes the ***setState*** method to update the state (the counter)

React.JS – Stateful Components

Counter Component…(cntd.)

- We use these methods as follows

```
class Counter extends React.Component {
        constructor(props, context) {
                super(props, context);

                this.state = {
                        seconds: 0
                };
        }

        render() {
                return (
                        <h1>{this.state.seconds}</h1>
                );
        }
}
```

Add this code after the constructor:

```
componentDidMount() {
        setInterval(this.timerTick, 1000);
}
timerTick() {
        this.setState({
                seconds: this.state.seconds + 1
        });
}
```

React.JS – Stateful Components

Counter Component…(cntd.)

- To bring in the stateful Component behaviour, that also ensures that state variable seconds is never out of sync

```
this.setState((prevState) => {
    return {
        seconds: prevState.seconds
        + 1
    };
});
```

```
timerTick() {
    this.setState({
        seconds: this.state.seconds + 1
    });
}
```

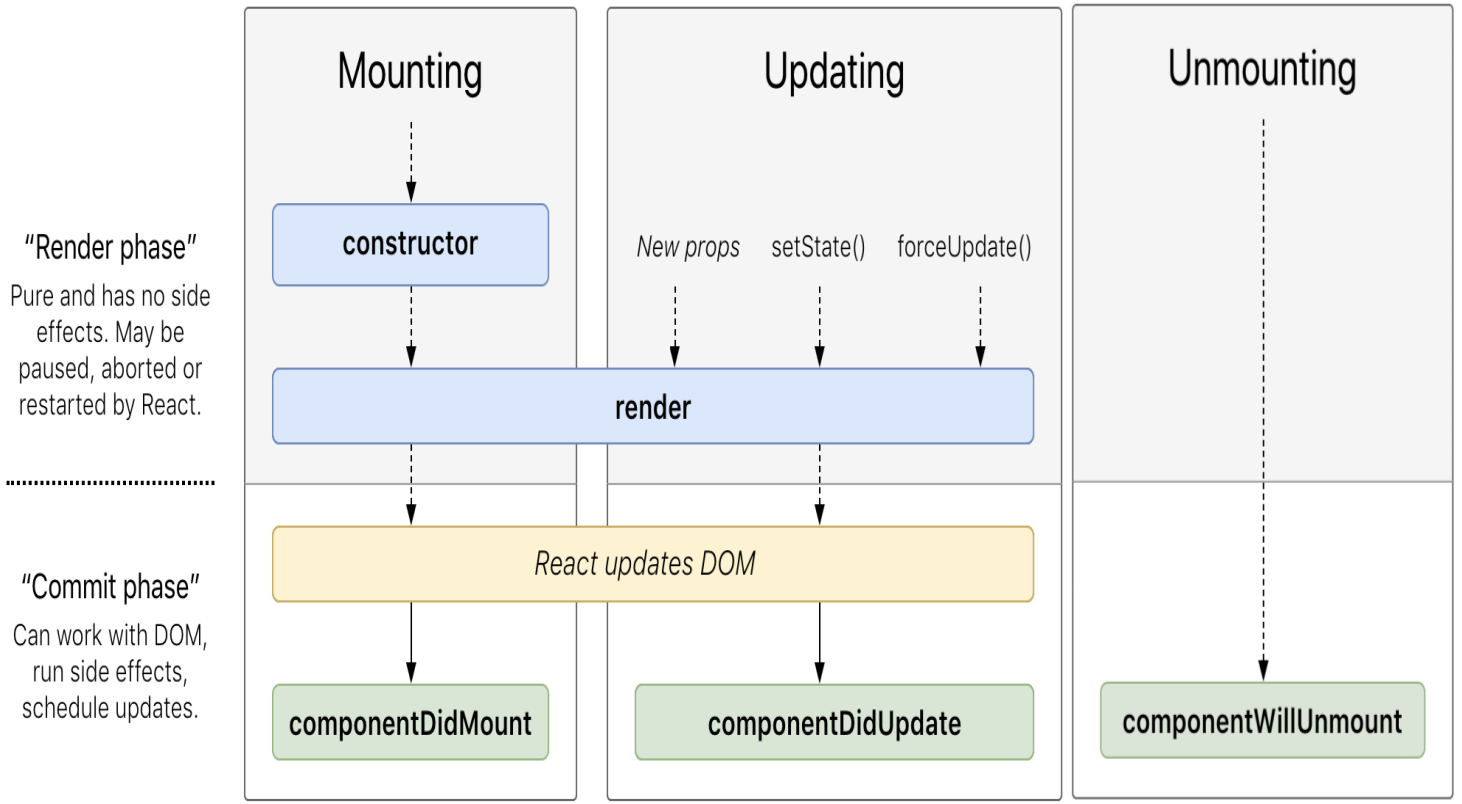
- To ensure that the this.state works well in the timerTick method, add the following line to call that function with the context of the Component

```
constructor(props, context) {
    …
    this.timerTick = this.timerTick.bind(this);
}
```

Component States and LifeCycle methods

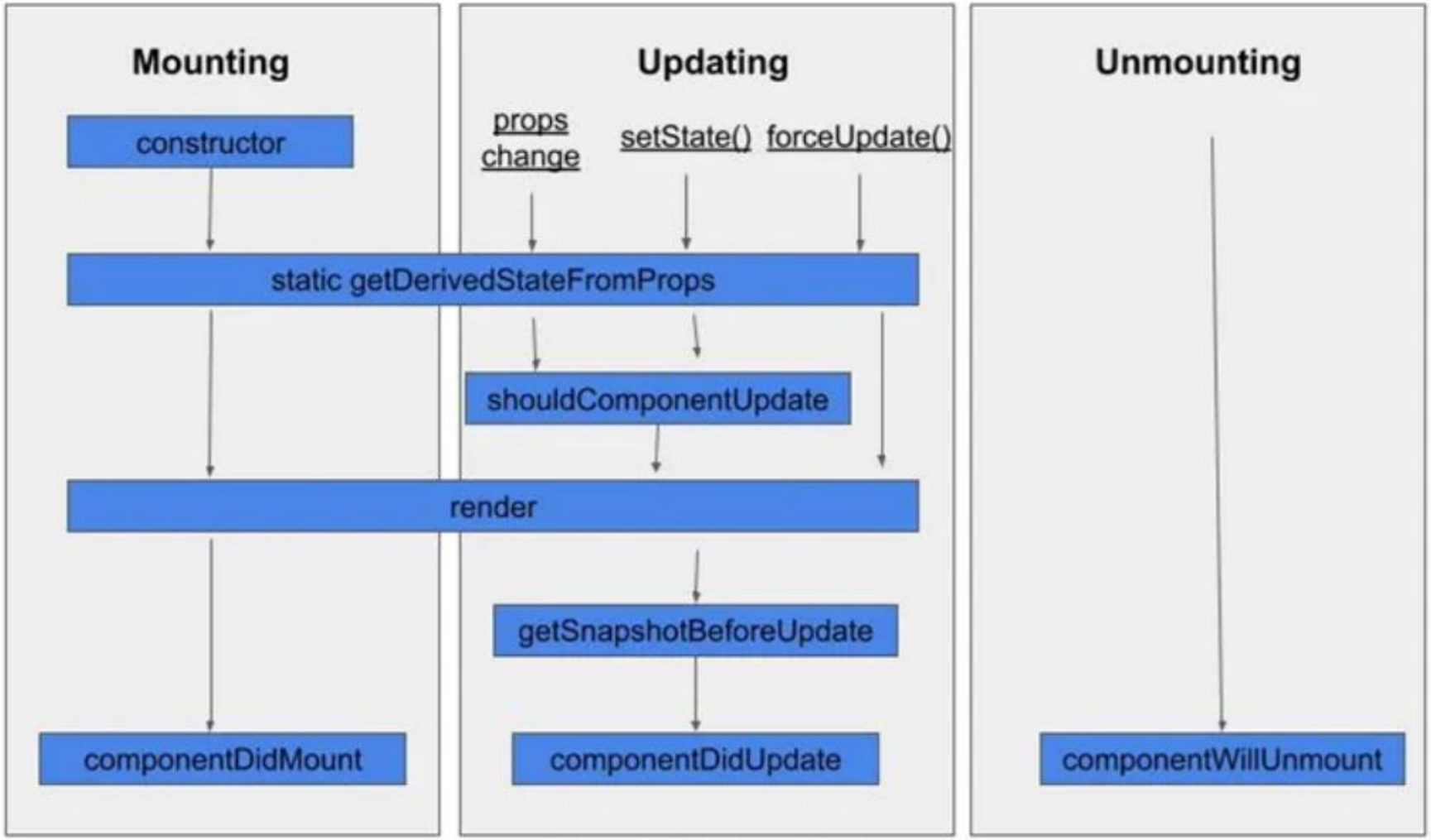
Introduction to Life Cycle

- The series of events that happen from the starting of a React component to its ending.
- Every component in React should go through the following lifecycle of events.
  - **Mounting** - Birth of the Component
  - **Updating**- Growing of component
  - **Unmounting**- End of the component



Component States and LifeCycle methods

Pictorial Representation of methods



Component States and LifeCycle methods

Methods

- **constructor()**
  - Called before the component is mounted.
  - Implementation requires calling of super(props) before further moving. Otherwise, this.props will be undefined in the constructor, which can lead to a major error in the application.
  - Supports Initializing the state and Binding our component
- **render()**
  - Most useful life cycle method as it is the only method that is required
  - Handles the rendering of component while accessing ***this.state*** and **this.props**

Component States and LifeCycle methods

Methods continued..

PES UNIVERSITY

- **componentDidMount()**
  - The best place to initiate API calls in order to fetch data from remote servers
  - Use setState which will cause another rendering but It will happen before the browser updates the UI. This is to ensure that the user won't see the intermediate state
  - AJAX requests and DOM or state updates should occur here
  - Also used for integration with other JavaScript frameworks like Node.js and any functions with late execution such as **setTimeout** or **setInterva**l

Component States and LifeCycle methods

Methods continued..

PES UNIVERSITY

getDerivedStateFromProps()

- This method is invoked right before rendering, both during the initial mount and when the component is re-rendered due to changes in props.
- It was used to compare the incoming props (nextProps) with the current props (this.props) and make state updates or logic adjustments accordingly before the next render.
- **shouldComponentUpdate()**
  - allows you to control whether a component should re-render when there are changes to its props or state.
  - Allows a component to exit the Update life cycle if there's no reason to use a replacement render.
  - It returns either true (the component will re-render) or false (the component will not re-render).

Component States and LifeCycle methods

Methods continued..

- **getSnapshotBeforeUpdate()-** for capturing information from the DOM before re-rendering. To capture some properties (a "snapshot") from the DOM before updates occur, which you can then pass into componentDidUpdate() to make adjustments after the update

- **componentDidUpdate()**
  - Is invoked immediately after updating occurs. Not called for the initial render.
  - Will not be invoked if **shouldcomponentUpdate() returns false**.
- **componentWillUnmount()**
  - Called when a component is being removed from the DOM

Component States and LifeCycle methods

Key Methods in the React Component Lifecycle: From Mounting to Unmounting

PES UNIVERSITY

Mounting (Birth of the Component): This phase occurs when a component is created and inserted into the DOM for the first time. The key lifecycle methods or hooks that run during this phase include:

- **constructor():** Initializes the component's state.
- **render():** Returns the JSX to display.
- **componentDidMount()** (class components) / **useEffect(() => {}, [])** (functional components): Executed after the component is mounted, typically used for API calls or subscriptions.

Component States and LifeCycle methods

Key Methods in the React Component Lifecycle: From Mounting to Unmounting

PES UNIVERSITY

Updating (Growth of the Component): This happens when the component's state or props change, causing it to re-render. The key methods/hooks during this phase include:

- **shouldComponentUpdate()** (class components): Determines whether the component should re-render.
- **render():** Re-renders the JSX when state or props change.
- **componentDidUpdate()** (class components) / **useEffect()** (functional components): Runs after the component has updated.

Component States and LifeCycle methods

Key Methods in the React Component Lifecycle: From Mounting to Unmounting

PES UNIVERSITY

Unmounting (End of the Component): This phase happens when the component is removed from the DOM. The key lifecycle method is:

- **componentWillUnmount()** (class components) / Cleanup function in **useEffect()** (functional components): Used for cleanup tasks like cancelling API calls or removing event listeners.

Component States and LifeCycle methods

Demo of diff lifecycle methods

PES UNIVERSITY

THANK YOU

Vinay Joshi and Dr.Sarasvathi V

Department of Computer Science and Engineering

vinayj@pes.edu

sarsvathiv@pes.edu

Acknowledgement

The slides are created from various internet resources with valuable contributions from multiple professors and teaching assistants in the university.

WEB TECHNOLOGIES

React JS – Stateless Components

Prof. Vinay Joshi and Dr. Sarasvathi V
Department of Computer Science and Engineering

Acknowledgement
The slides are created from various internet resources with valuable contributions from multiple professors and teaching assistants in the university.

Stateless Components

Agenda

PES UNIVERSITY

- Introduction
- Demo
- Key differences

Stateless Components

Introduction

PES UNIVERSITY

- A component that has **no internal state management** in it
- Can be **written as functions that just return the JSX** element
- Simple **functional components without having a local state**
- Usually associated with how a concept is presented to the user
- Properties are passed like regular parameters. The props are displayed like **{props.name}**
- Hook in react is available to add state behavior in functional component

Stateless Components

Demo

- Sample code

```
function Demo(props) {
  return <h1> Welcome to REACT JS, {props.Name} </h1>;  }
```

- Code to render such components remains the same.

```
ReactDOM.render(<Demo Name = “friend”/>, destination )
```

```
//Rendering the component in React 18
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Demo Name="friend" />);
```

Stateless Components

Demo

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>React 18 Example</title>
</head>
<body>
   <div id="root"></div>
   <script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin></script>
   <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" crossorigin></script>
   <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
   <script type="text/babel">
      function Demo(props) {
         return <h1>Welcome to React JS, {props.Name}</h1>;
      }
      const root = ReactDOM.createRoot(document.getElementById('root'));
      root.render(<Demo Name="friend" />);
   </script>
</body>
</html>
```

Welcome to React JS, friend

Stateless Components

Key differences - Stateless and Stateful components

- **Stateless component**
  - **Either functional components** *(the standard approach today)* **or class component**
  - It takes an input (props) and returns the output (react element)
  - No internal state management in it
  - Also known as Presentational components/Dump components

- **Stateful component**
  - **Always a class component**(traditional approach) *or a functional component using React Hooks.*
  - It is created by extending the React.Component class.
  - Dependent on its state object and can change its own state. The component re-renders based on changes to its state
  - Also known as Smart or container Components

THANK YOU

Vinay Joshi and Dr.Sarasvathi V

Department of Computer Science and Engineering

vinayj@pes.edu

sarsvathiv@pes.edu

Acknowledgement

The slides are created from various internet resources with valuable contributions from multiple professors and teaching assistants in the university.

WEB TECHNOLOGIES

React JS – refs

Prof. Vinay Joshi and Dr. Sarasvathi V
Department of Computer Science and Engineering

Acknowledgement
The slides are created from various internet resources with valuable contributions from multiple professors and teaching assistants in the university.

refs and keys

Agenda

- Introduction to refs
- Create and use refs
- Callback refs

refs and keys

Introduction to refs

PES UNIVERSITY

- Provides a **way to directly access and interact with DOM nodes or React elements created in the render method**

- Used to return a reference to DOM node or React element, allowing us to perform operations directly on them

- Refs are generally used when we need to perform actions that are not easily achieved through declarative React patterns.

Introduction to refs

- Good use cases for refs to be used:
  - **Managing Focus, Text Selection, or Media Playback**: To programmatically manage focus, text selection, or control media elements.
  - **Triggering Imperative Animations**: Refs can be used to trigger animations that require direct manipulation of the DOM.
  - **Integrating with Third-Party DOM Libraries**: When integrating with libraries that interact with the DOM, refs provide a way to access and manipulate the DOM elements.

refs and keys

How to create and use refs?

PES UNIVERSITY

- **Creating Refs:**
  - **React.createRef()** is used to create refs in React, and it's typically done in the constructor (or with a class field).
  - These refs can then be attached to a React element via the ref attribute.

- **Accessing Refs:**
  - After the ref is attached to an element in the render() method, the DOM node can be accessed via **this.myRef.current**.
  - This is useful when we need to directly interact with the DOM node, like focusing on an input element or retrieving its value.

refs and keys

How to create and use refs? (coding example)

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    // Creating a ref using React.createRef()
    this.myRef = React.createRef();
  }

  componentDidMount() {
    // Accessing the DOM node when the component mounts
    const node = this.myRef.current;
    console.log('DOM Node:', node);
  }

  render() {
    return (
      // Attaching the ref to the div element
      <div ref={this.myRef} style={{ width: '100px', height: '100px', backgroundColor: 'lightblue' }}>
        Ref Demo
      </div>
    );
  }
}
```


```
// Rendering the component
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyComponent />);
```

refs and keys

Callback refs

PES UNIVERSITY

Fine-Grain Control:

- Callback refs give more control over when refs are set and unset. We can define a function that will be called when the element is rendered or removed.

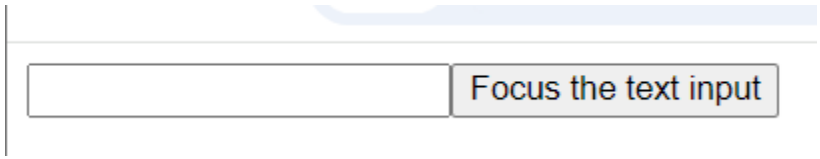
Function as Ref:

- Instead of using React.createRef(), a function is used as the ref to assign the DOM element or component instance to a class property.
- This function receives the React component instance or HTML DOM element as an argument, allowing us to store and access the ref elsewhere.

refs and keys

Callback refs (coding example)

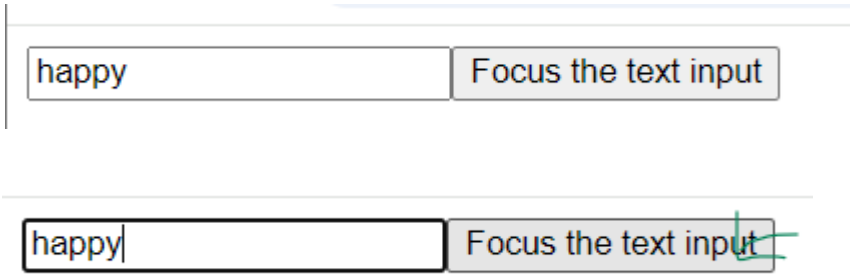
```
class CustomTextInput extends React.Component {

  constructor(props) {

    super(props);

    this.textInput = null;


    // Callback ref function to assign the DOM element to this.textInput

    this.setTextInputRef = element => {

      this.textInput = element;

    };


    this.focusTextInput = () => {

      // Use the raw DOM API to focus the text input if it exists

      if (this.textInput) this.textInput.focus();

    };

  }
```



refs and keys

Callback refs (coding example) continuation…

```
componentDidMount() {
  // Automatically focus the input when the component mounts
  this.focusTextInput();
}

render() {
  return (
    <div>
      {/* Use the callback ref to get the input DOM element */}
      <input type="text" ref={this.setTextInputRef} />
      <input type="button" value="Focus the text input" onClick={this.focusTextInput} />
    </div>
  );
}}
```



THANK YOU

Vinay Joshi and Sarasvathi V

Department of Computer Science and Engineering

vinayj@pes.edu

sarsvathiv@pes.edu

Acknowledgement

The slides are created from various internet resources with valuable contributions from multiple professors and teaching assistants in the university.

WEB TECHNOLOGIES

React JS – Keys

Prof. Vinay Joshi and Prof. Sindhu R Pai
Department of Computer Science and Engineering

Acknowledgement
The slides are created from various internet resources with valuable contributions from multiple professors and teaching assistants in the university.

refs and keys

Agenda

- Introduction to keys
- Keys - Sample code
- Map function
- Using map with key property

Keys

Introduction to keys

- Keys allow React to keep track of elements.
- This way, if an item is updated or removed, only that item will be re-rendered instead of the entire list.

Warning: Each child in an array or iterator should have a unique "key" prop.

- Keys need to be unique to each sibling.
- Special string attribute while creating list of elements in React
- Utilized to identify **specific virtual DOM elements** that have changed, added, or removed
- Generally, the key should be a unique ID assigned to each item.
- We can use the array index as a key.
- Not specifying the key property will display a warning on the console:

refs and keys

Keys - Sample codes

PES UNIVERSITY

Using keys in a list of elements :

```
function SuperheroList() {
    const superheroes = [
        { id: 1, name: 'Batman' },
        { id: 2, name: 'Ironman' },
        { id: 3, name: 'Spiderman' }
    ];

    return (
        <div>
            {superheroes.map(hero => (
                <p key={hero.id}>{hero.name}</p> // Using 'id' as a unique key
            ))}
        </div>
    );
}

export default SuperheroList;
```

Specifying keys directly:

```
import React from 'react';

function SuperheroList() {
    return (
        [
            <p key="1">Batman</p>,
            <p key="2">Ironman</p>,
            <p key="3">Spiderman</p>
        ]
    );
}

export default SuperheroList;
```

refs and keys

Keys - Sample codes

```
// Example usage
const todos = [
  { id: 1, text: 'Learn React' },
  { id: 2, text: 'Build a Project' },
  { id: 3, text: 'Review Concepts' },
];

// Rendering the TodoList component
const App = () => {
  return (
    <div>
      <h1>Todo List</h1>
      <TodoList todos={todos} />
    </div>
  );
};

export default App;
```

refs and keys

Map function

- Map is a data collection type where data is stored in the form of key-value pairs
- The value stored in the map must be mapped to the key
- The map is a JavaScript function that can be called on any array
- With the map function, we map every element of the array to the custom components in a single line of code
- Sample code:

```
const numbers = [1, 2, 3, 4, 5];

const doubled = numbers.map((number) => number * 2);

console.log(doubled);
```

refs and keys

Using map with key property

- The provided code snippet will result in warning in react because the list items generated from the numbers arrays do not have a unique key property.
- React uses these keys to identify which items have changed, are added, or are removed, and not having keys can lead to inefficiencies and incorrect behavior when rendering lists.

```
import React from 'react';
import ReactDOM from 'react-dom';

const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) => <li key={number}>{number}</li>);

ReactDOM.render(
    <ul>{listItems}</ul>,
    document.getElementById('root')
);
```

refs and keys

Using map with key property

PES UNIVERSITY

- Modified by **adding key property**(The modification primarily revolves around the proper use of **keys** in list rendering)

```
function NumberList(props) {
    const numbers = props.numbers; // Destructuring the numbers prop
    const listItems = numbers.map((number) =>
        <li key={number.toString()}> {number} </li>
    );
    return (
        <ul>{listItems}</ul> // Returning an unordered list containing the list items
    );
}

const numbers = [1, 2, 3, 4, 5]; // Array of numbers

ReactDOM.render(
    <NumberList numbers={numbers} />, // Rendering the NumberList component
    document.getElementById('root') // Targeting the root element in the HTML
);
```

THANK YOU

Vinay Joshi and Sindhu R Pai
Department of Computer Science and Engineering

vinayj@pes.edu
+91 9886703973

sindhurpai@pes.edu
+91 8277606459

WEB TECHNOLOGIES

React JS – Event Handling

Prof. Sindhu R Pai
Department of Computer Science and Engineering

ReactJS - Event Handling

Agenda

PES UNIVERSITY

- Introduction
- Synthetic Event objects
- Requirement
- Event Registration
- Demo
- Programming assignments

ReactJS - Event Handling

Introduction

- Events make the **web app interactive and responsive** to the user.
- React event handling system is known as **Synthetic Events.**
- Almost same as the DOM events. A few differences in the syntax
  - With JSX in **ReactJs, you pass a function as event handler** and in **DOM element we pass function as string**

```
// event handling in ReactJs element
<input id="inp" name="name" onChange={onChangeName} />
```

```
// event handling in DOM element
<input id="inp" name="name" onchange="onChangeName()"/>
```

  - In **DOM elements the event name is in lowercase** while in **ReactJs it is in camalCase**.
  - In ReactJs, **you can not prevent default behavior by returning false from the event**. In ReactJs, you need to explicitly call preventDefault of the event.

ReactJS - Event Handling

Synthetic Event objects

- The event object passed to the event handlers are **SyntheticEvent object**
- **A wrapper** around the DOMEvent object
- The event handlers are registered at the time of rendering
- Returning false does not prevent the default browser, use e.preventDefault() or e.stopPropogation() accordingly
- **Properties**
  - boolean bubbles
  - boolean cancelable
  - DOMEventTarget currentTarget
  - boolean defaultPrevented
  - number eventPhase
  - boolean isTrusted
  - DOMEvent nativeEvent
  - string type

ReactJS - Event Handling

Requirement

- Clicking on the + button, the value of our counter must be incremented by one
- Involves:
  - Listen for the click event on the button
  - Implement the event handler where we react to the click and increase the value of **this.state.count** that counter relies on

ReactJS - Event Handling

Event Registration

- In React, you listen to an event by specifying everything inline in your JSX itself
- Specify both the **event** you are listening for and the **event handler** that will get called **inside the markup**

ReactJS - Event Handling

Demo

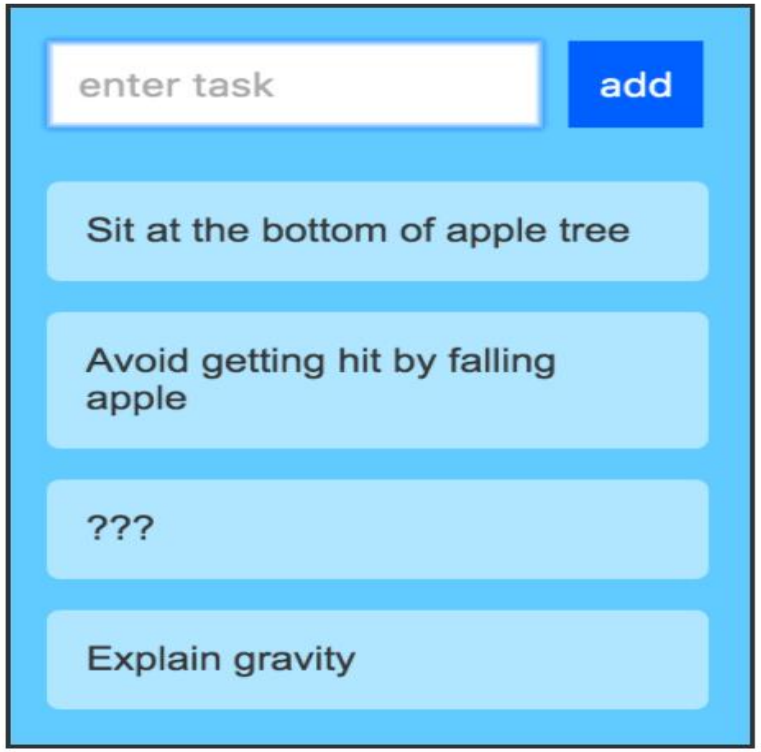
PES UNIVERSITY

- Demo of event handling

ReactJS - Event Handling

Programming assignment

- Build an awesome todo list as shown.



THANK YOU

Sindhu R Pai
Department of Computer Science and Engineering

sindhurpai@pes.edu
+91 8277606459

www.kirupa.com

WEB TECHNOLOGIES

React.JS – Forms

Vinay Joshi
Department of
Computer Science and Engineering

React.JS – Forms

Introduction

PES UNIVERSITY

- Two main functionalities associated with any form is when
  - input values are changed (using ***onChange*** event)
  - form is submitted (using ***onSubmit*** event)
- Form Data in React is usually handled by Components by storing them in ***state*** object, such Form components are called ***Controlled Components***

React.JS – Forms

Controlled Components

- The value property of the three types of form elements <input>, <textarea> and <select> are controlled by React using the ***state*** and updated only using ***setState***
- The value is updated in the ***state*** when onChange event is triggered on the form element (by setState)
- The value is also set to the state property to keep it updated at all times (updated by React) – this is termed as ***"single source of truth"***
- one authoritative location (source) stores and maintains a piece of data or information

React.JS – Forms
Controlled Components

- <textarea> Default Value </textarea>

  can be changed to

  <textarea value={this.state.value} />
- <select>

  <option select value="optionselected"> Default Value </option>

  </select>

  can be changed to

  <select value={this.state.value}>

  <option value="optionselected">Default Value</option>

  </select>

React.JS – Forms

Handling Multiple Inputs

- To handle multiple inputs by writing a common change handler as follows

```
handleChange(event) {
        name = event.target.name;
        value = event.target.value;
        this.setState({
                [name]: value
        });
```

where [name] is the notation for computed property name

Note: setState can be called only the property that is changed

React.JS – Forms

Uncontrolled Components

- To write an uncontrolled component, instead of writing an event handler for every state update, you can use a ref to get form values from the DOM.
- This we have already seen in the previous lessons
- Additionally, use the defaultValue property to specify initial value in React

```
<input defaultValue="Bob" type="text" ref={this.input} />
```

THANK YOU

Vinay Joshi
Department of Computer Science and Engineering

vinayj@pes.edu

+91 80 2672 6622

WEB TECHNOLOGIES

React Hooks

Prof. Pavan. A. C

Department of Computer Science and Engineering

React Hooks

- Hooks allow us to "hook" into React features such as state and lifecycle methods.
- Hooks allow function components to have access to state
- There are 3 rules for hooks:
    - Hooks can only be called inside React function components.
    - Hooks can only be called at the top level of a component.
    - Hooks cannot be conditional
- Hooks will not work in React class components.

React Hooks

React useState Hook

PES UNIVERSITY

- You need to include Hooks from the React library if you are using React APP.
- The React useState Hook allows us to track state in a function component.
- We initialize our state by calling useState in our function component.
- useState accepts an initial state and returns two values:
  - The current state.
  - A function that updates the state.

React Hooks

React useState Hook

```
function FavoriteColor() {
        const [color, setColor] = React.useState("");
}
```

- We are destructuring the returned values from useState.
  - The first value, color, is our current state.
  - The second value, setColor, is the function that is used to update our state.

React Hooks

React useEffect Hook

- useEffect Hook allows you to perform side effects in your components.
- useEffect accepts two arguments. The second argument is optional.

  useEffect(<function>, <dependency>)
- If no dependencies are passed then rendering will happen continuously.
- If an empty array is passed as dependency the it runs only on the first render.
- If props or state values are given then render will be called and dependency values changes.

THANK YOU

Prof. Pavan A C

Department of Computer Science and Engineering

pavanac@pes.edu | +91-9481187128

NODE JS

Revathi G P
Department of
Computer Science and Engineering

NODE JS

NodeJS Introduction

Revathi G P
Department of Computer Science and Engineering

NodeJS Introduction

- Node.js is an open source, cross-platform runtime environment built on Google Chrome's JavaScript Engine (V8 Engine).
- Node.js was developed by Ryan Dahl in 2009 and its latest version is v20.17.0
- 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 mac 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**

Why Node?

When you create websites with PHP for example, you associate the language with an HTTP web server such as Apache or Nginx. Each of them has its own role in the process:

- Apache manages HTTP requests to connect to the server. Its role is more or less to manage the in/out traffic.
- PHP runs the .php file code and sends the result to Apache, which then sends it to the visitor.

As several visitors can request a page from the server at the same time, Apache is responsible for spreading them out and running different *threads* at the same time.

Each thread uses a different processor on the server (or a processor core)

Why Node?

- A Node.js app runs in a single process, without creating a new thread for every request.
- Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.
- When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

NODE JS

Why Node?

PES UNIVERSITY

A common task for a web server can be to open a file on the server and return the content to the client.

Here is how PHP or ASP handles a file request:

- Sends the task to the computer's file system.
- Waits while the file system opens and reads the file.
- Returns the content to the client.
- Ready to handle the next request.

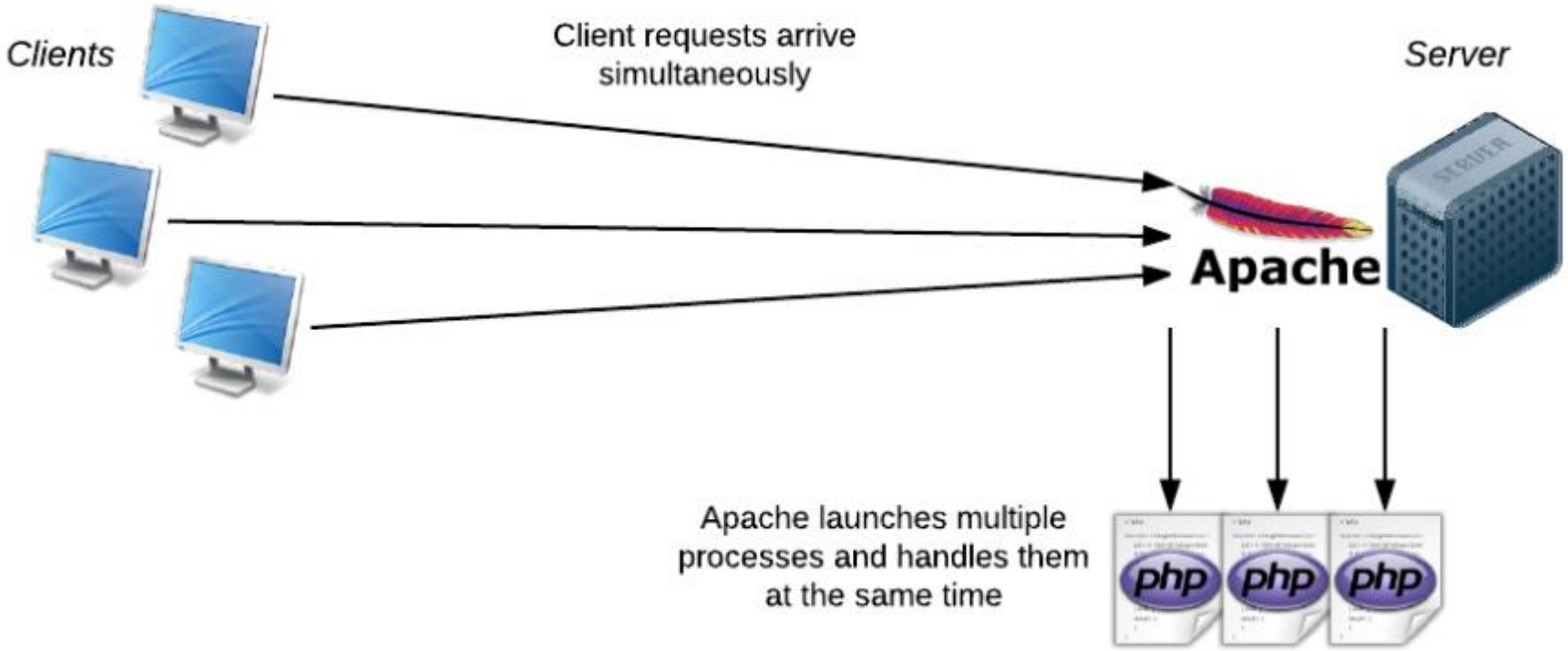
Here is how Node.js handles a file request:

- Sends the task to the computer's file system.
- Ready to handle the next request.
- When the file system has opened and read the file, the server returns the content to the client.
- Node.js eliminates the waiting, and simply continues with the next request.

Node.js runs single-threaded, non-blocking, asynchronously programming, which is very memory efficient.

NODE JS

Why Node?



The Apache server is multithread

NODE JS

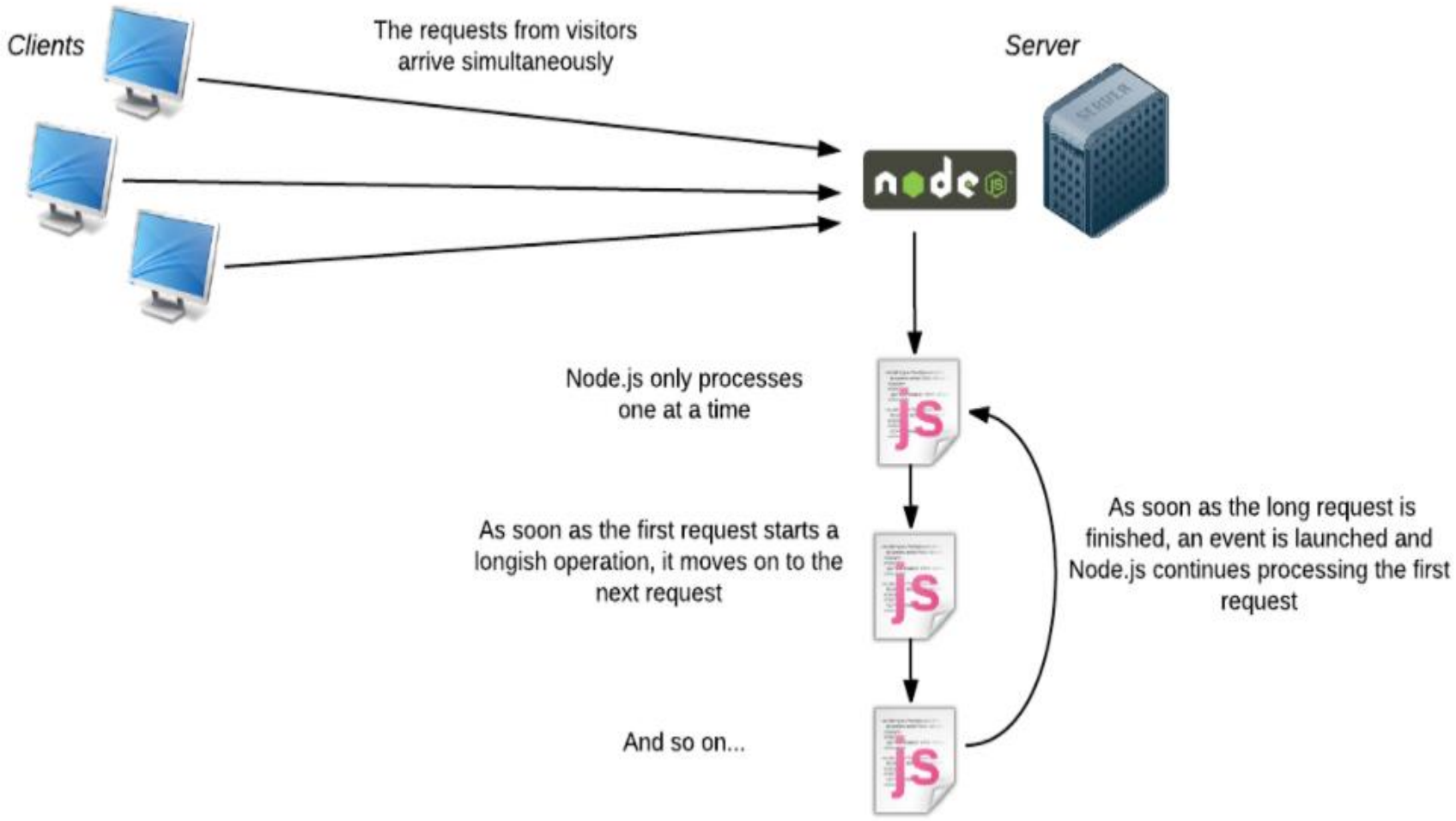
Mono-Thread – Handling multiple Requests?

- Node.js doesn't use an HTTP server like Apache. In fact, it's up to us to create the server! Isn't that great?
- Unlike Apache, Node.js is **monothread**. This means that there is only one process and one version of the program that can be used at any one time in its memory.

But I thought that Node.js was really fast because it could manage loads of requests simultaneously. If it's monothread, can it only perform one action at a time?

NODE JS

Mono-Thread Style



NODE JS

V8 Engine and C++ Bindings

PES UNIVERSITY

- 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.

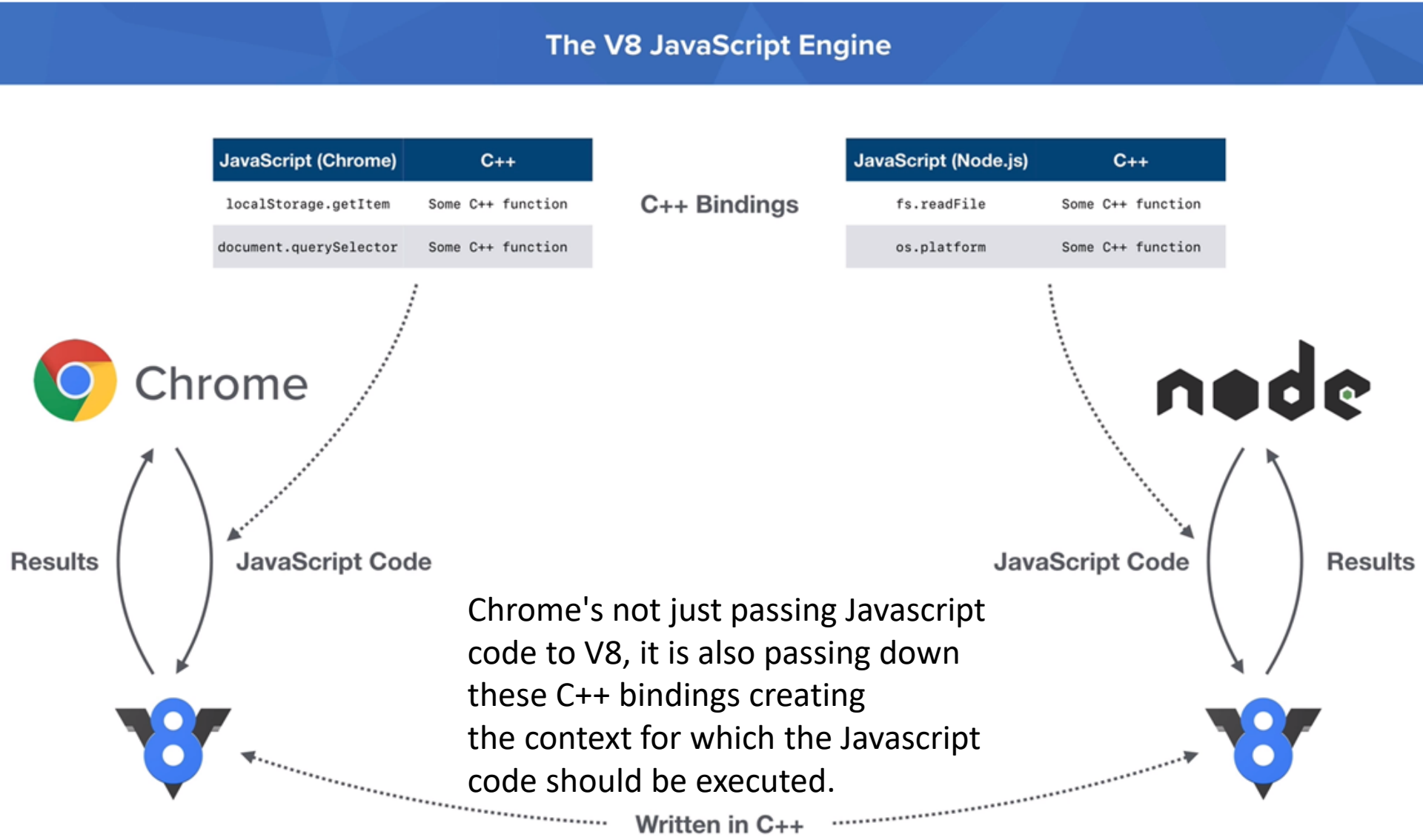
Example Instance:

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.

PES UNIVERSITY

What Can Node.js Do?

- Node.js can generate dynamic page content
- Node.js can create, open, read, write, delete, and close files on the server
- Node.js can collect form data
- Node.js can add, delete, modify data in your database

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

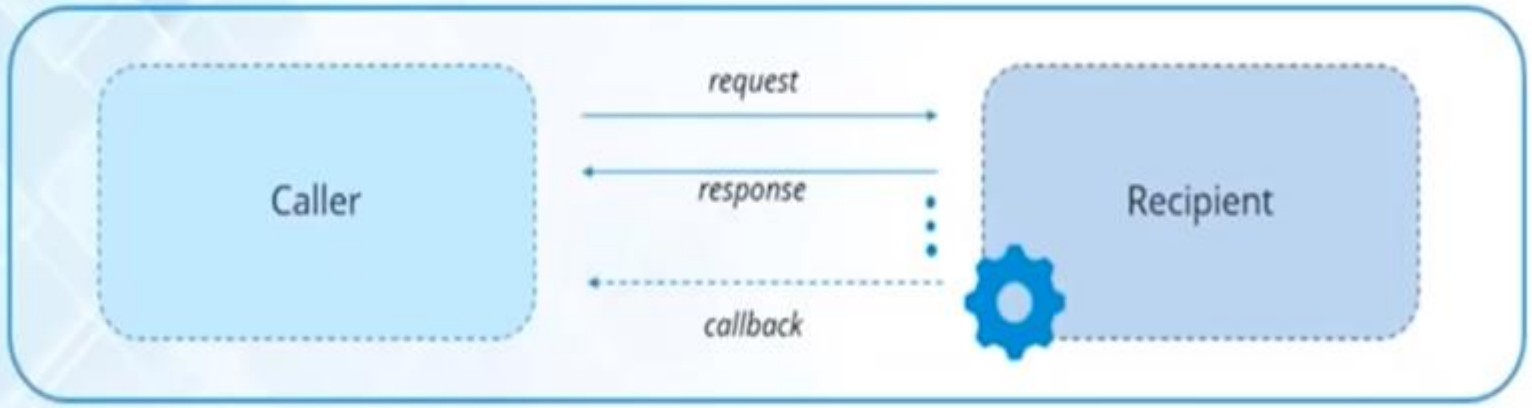
Node.js applications never buffer any data. These applications simply output the data in chunks.

NODE JS

Asynchronous and Event Driven

PES UNIVERSITY

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

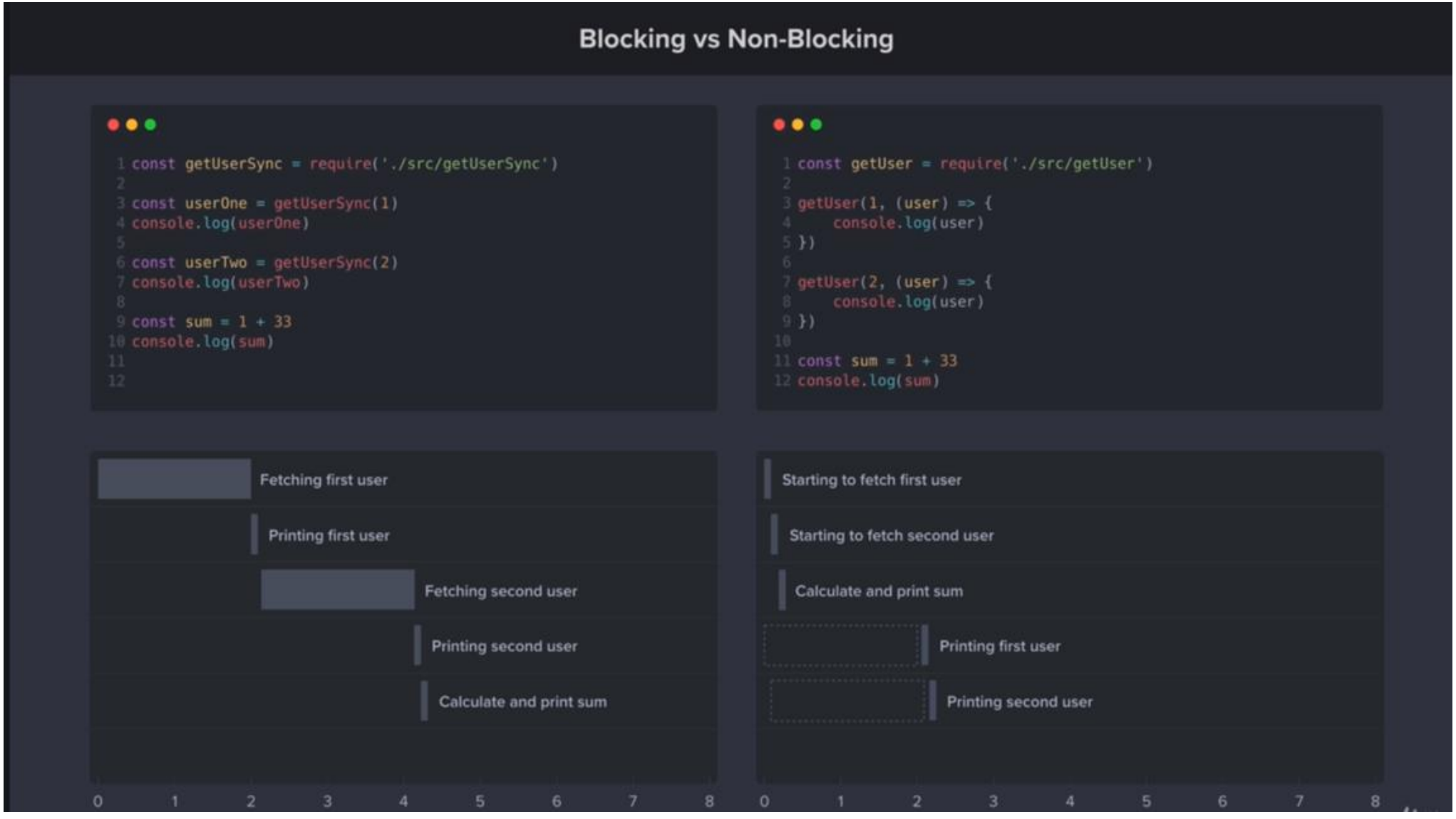
PES UNIVERSITY

- 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

Non- Blocking I/O



NODE JS

Blocking and Non- Blocking I/O

PES UNIVERSITY

BLOCKING I/O

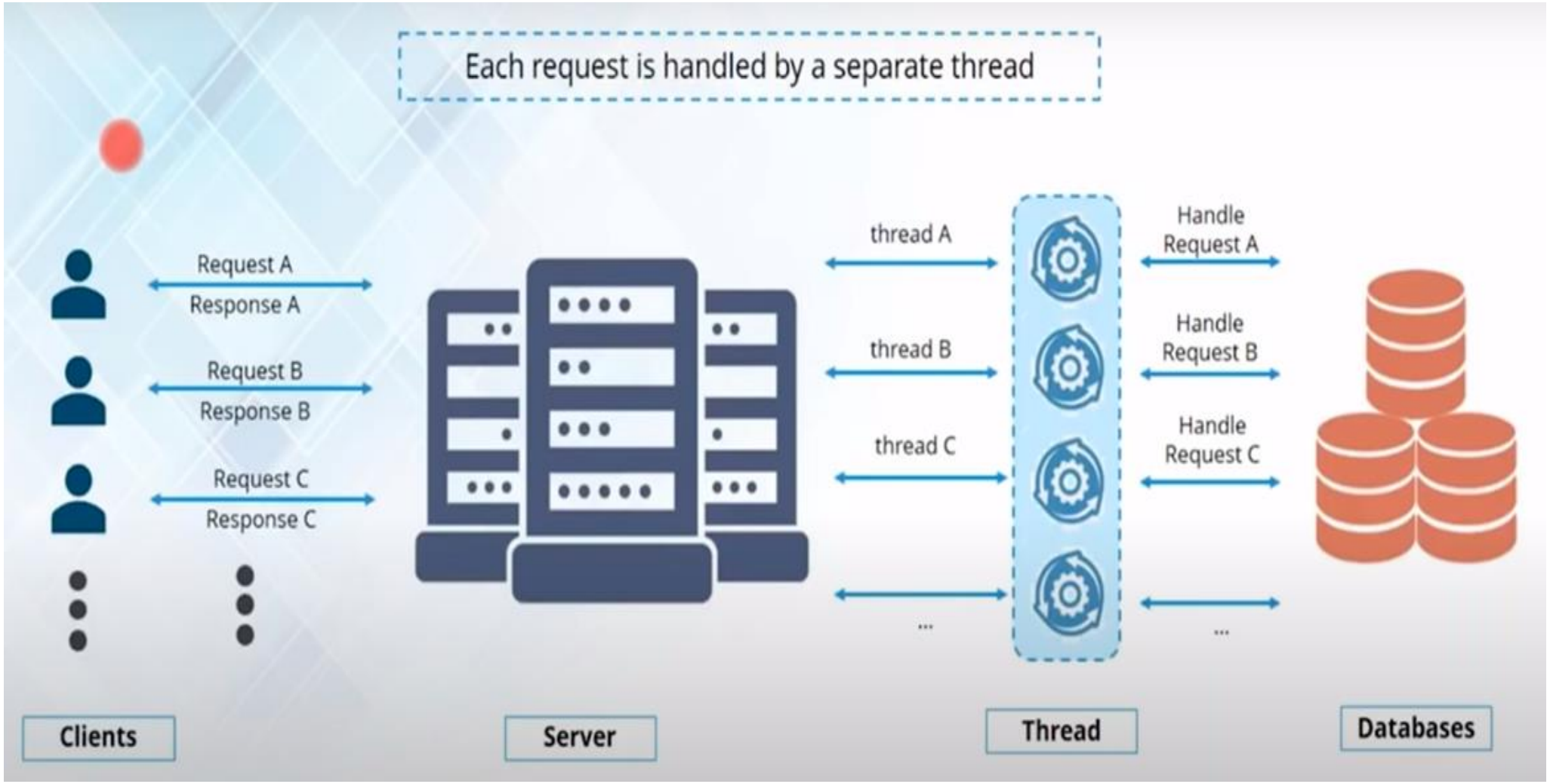
```
var fs = require('fs');
var data = fs.readFileSync('test.txt');
console.log(data.toString());
console.log('End here');
```

```
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

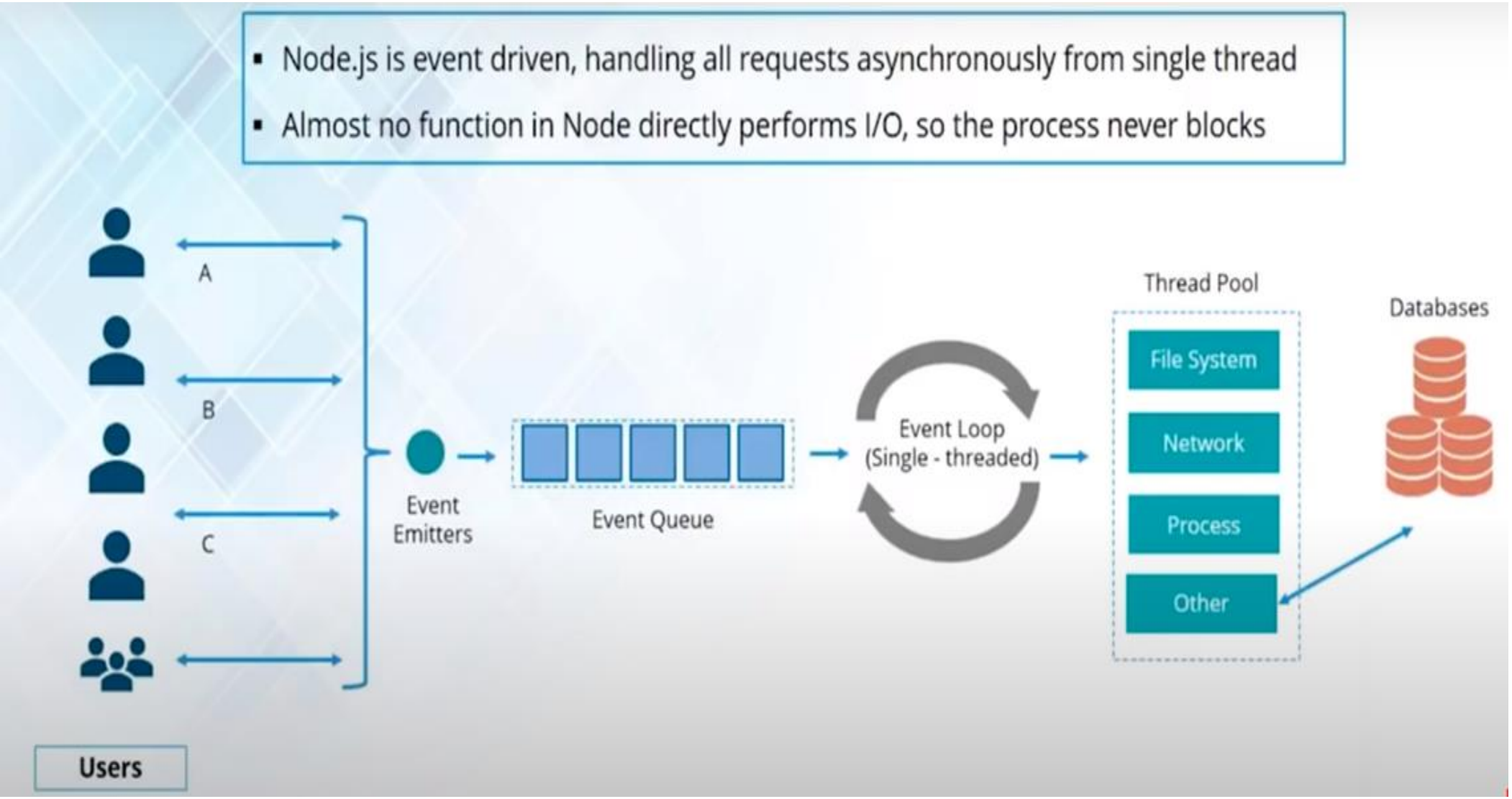
NODE JS

Multi Threaded Model



NODE JS

Single Threaded Model



NODE JS

Multi Threaded vs Asynchronous Event Driven Model

PES UNIVERSITY

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.

Link: https://www.youtube.com/watch?v=8aGhZQkoFbQ

NODE JS

Node JS Success Stories

UBER Story

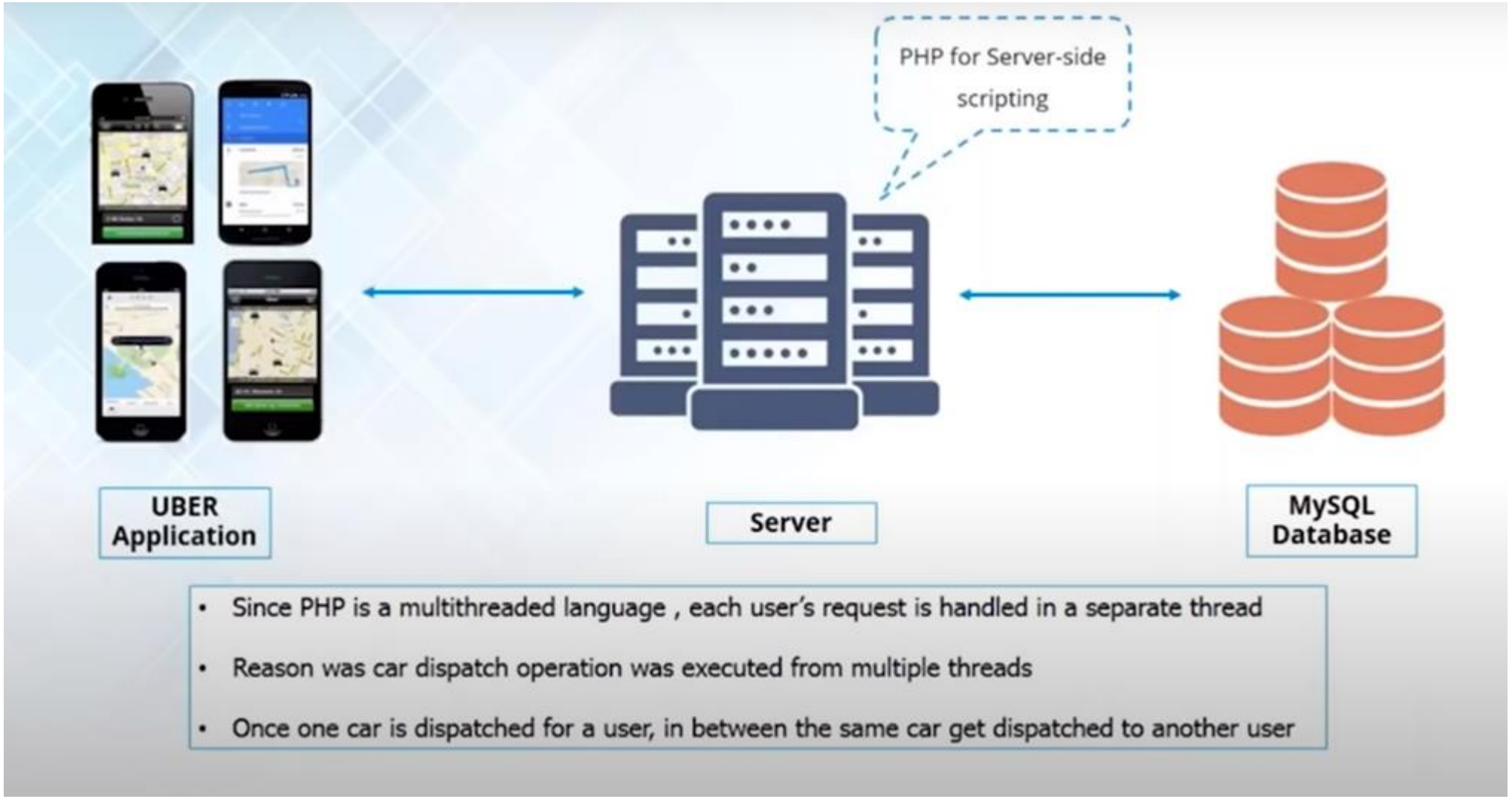
- Customers interact with Uber applications and generating request to book a cab.
- Requests are sent to the Uber server to check in the geospatial databases for the cab.
- Server send back the card details or driver details back to the customer in form of response.

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

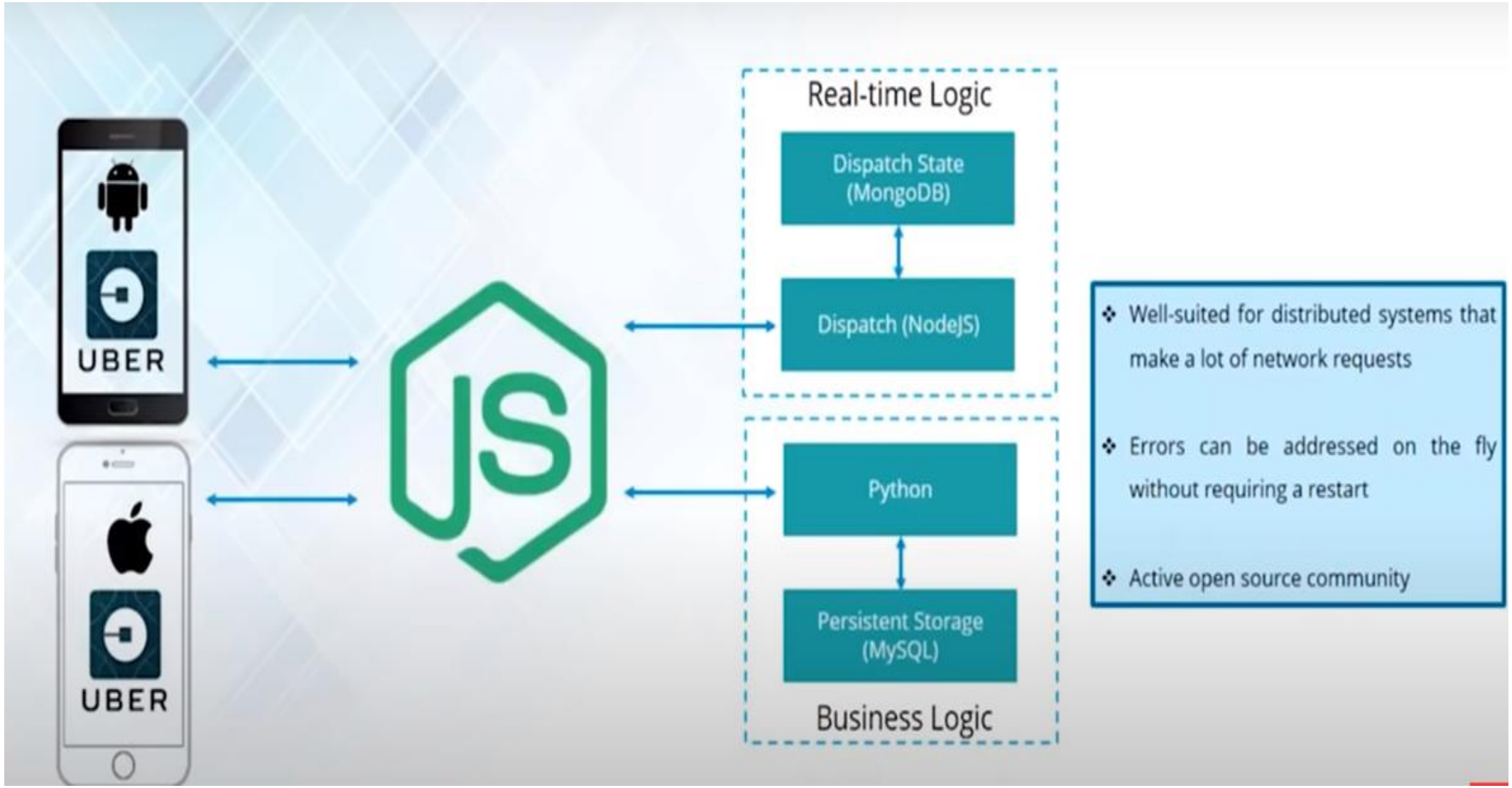
UBER Story – Old Architecture



- 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

PES UNIVERSITY

- 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

THANK YOU

Revathi G P
Department of
Computer Science and Engineering
revathigp@pes.edu

NODE JS: Set up Node JS app

Revathi G P

Department of Computer Science and Engineering

Acknowledgement

The slides are created from various internet resources with valuable contributions from multiple professors and teaching assistants in the university.

Node JS installation

To install and setup an environment for Node.js, you need the following two softwares available on your computer:

Text Editor.
Node.js Binary installable

Node JS installation

The Node.js Runtime:

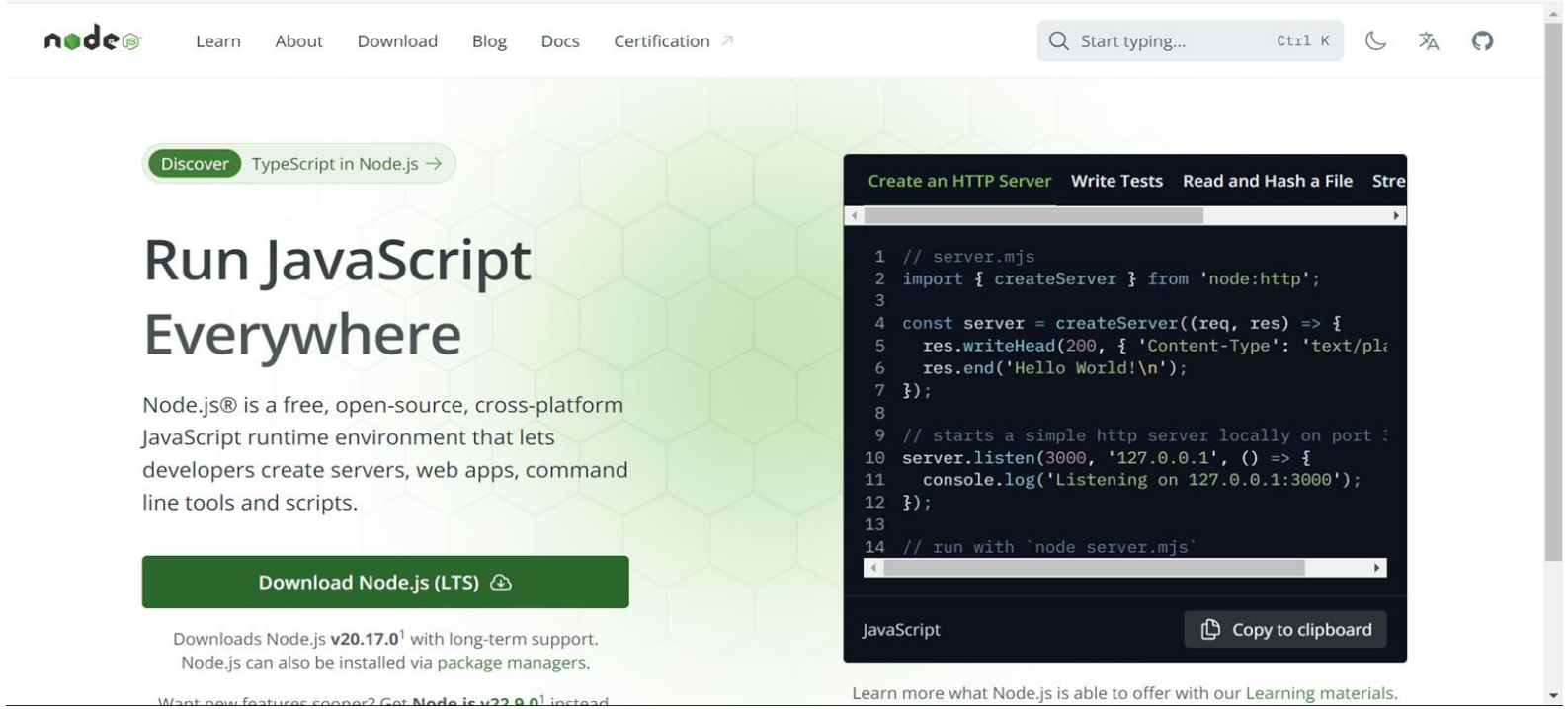
- The source code written in source file is simply JavaScript.
- It is interpreted and executed by the Node.js interpreter.

NODE JS

Node JS installation

How to download Node.js:

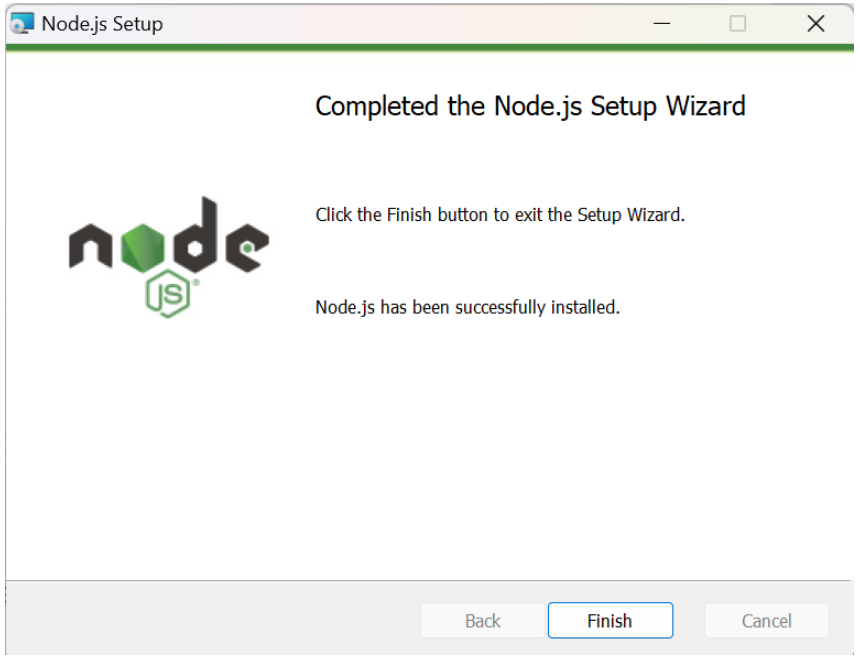
You can download the latest version of Node.js installable archive file from https://nodejs.org/en/



NODE JS

Node JS installation

- Step 1: **Go to the NodeJS website and download NodeJS**
  - Go to the NodeJS website https://nodejs.org/en/ and download NodeJS.
  - Look for the Latest Stable Version and download it



Node JS installation

Step 2: Make sure Node and NPM are installed and their PATHs defined

After Installing, Verify the installation by giving the command as

node –v //should return the version number

Npm –v // should return the version number of npm package

Returns 'node' is not recognized as an internal or external command, operable program or batch file. Nodejs not installed properly check for path.

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

PES UNIVERSITY

Step 3: Updating the Local npm version.

The final step in node.js installed is the updation of your local npm version(if required) – the package manager that comes bundled with Node.js.

You can run the following command, to quickly update the npm

npm install npm –global // Updates the ‘CLI’ client

NODE JS

Node JS installation

Node.js First Example

There can be console-based and web-based node.js applications.

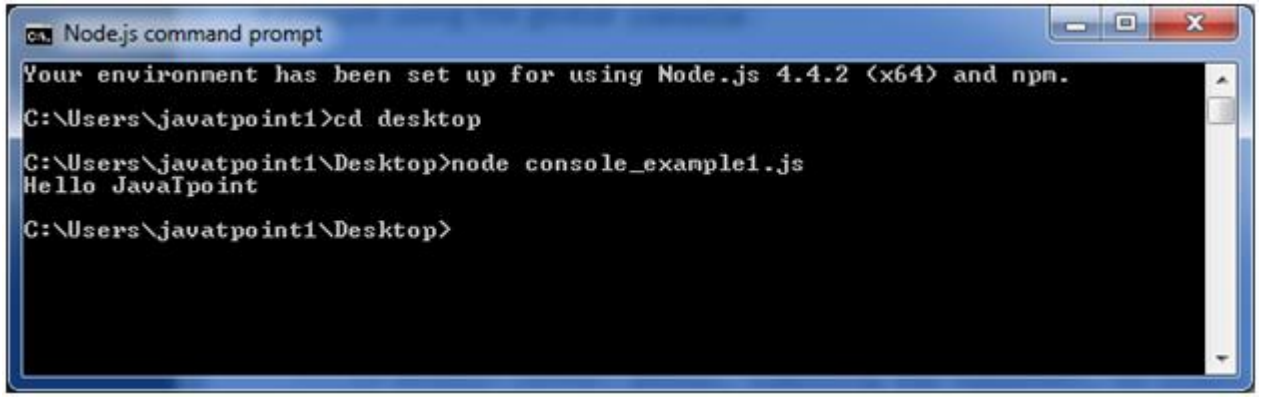
Node.js console-based Example

File: console_example1.js

```
console.log('Hello World');
```

Open Node.js command prompt and run the following code:

node console_example1.js



NODE JS

Node JS installation

web-based node.js applications.

Once you have downloaded and installed Node.js on your computer, let's try to display "Hello World" in a web browser.

Create a Node.js file named "myfirst.js", and add the following code:

myfirst.js

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

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end('Hello World!');
}).listen(8080);
```

Save the file on your computer: C:\Users\Your Name\myfirst.js

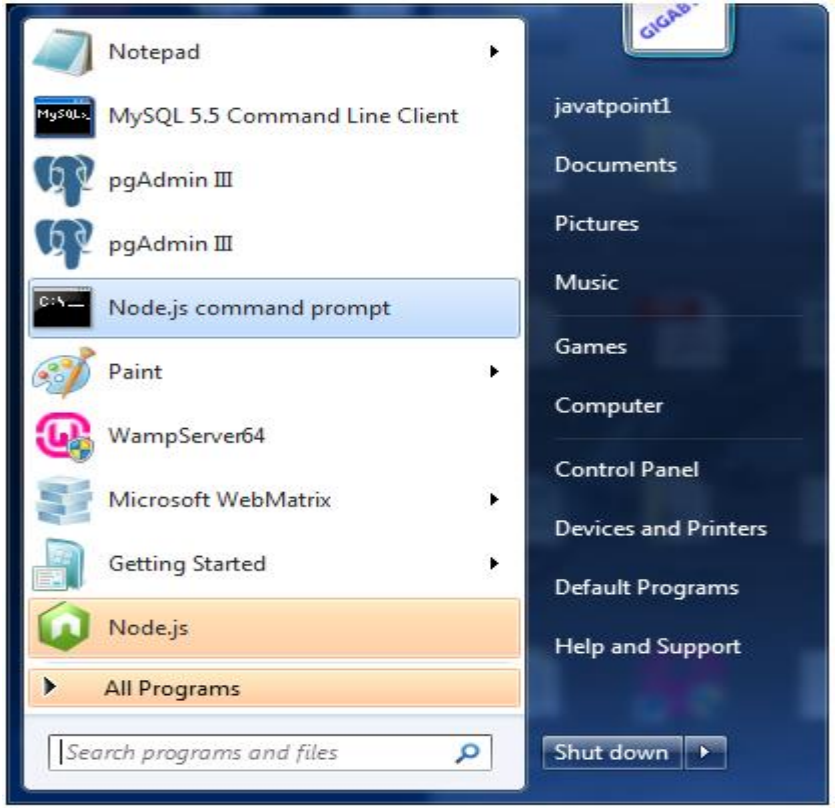
The code tells the computer to write "Hello World!" if anyone (e.g. a web browser) tries to access your computer on port 8080.

NODE JS

Node JS installation

How to start your server:

Go to start menu and click on the Node.js command prompt.



Initiate the Node.js File
The file you have just created must be initiated by Node.js before any action can take place.

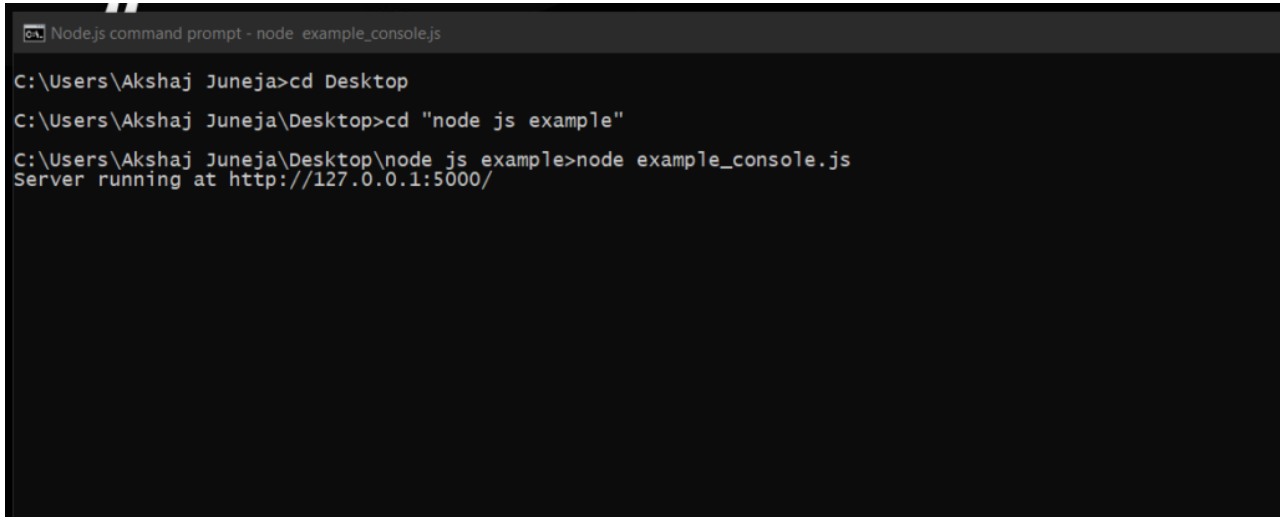
Start your command line interface, write node myfirst.js

NODE JS

Node JS installation

To run this myfirst.js file follow the steps as given below:

Search the node.js command prompt in the search bar and open the node.js command prompt.
Go to the folder using cd command in command prompt and write the following command node web.js



Node JS installation

- Step 3: **Create a New Project Folder**
- Step 4: **Start running NPM in your project folder**
  - open a terminal.
  - change directories until you are in your project folder.
  - run the command npm init in the terminal

```
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (simple-node-server)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author: Kris Hill
license: (ISC)
About to write to C:\Users\krist\OneDrive\Documents\Blog Posts\Example Projects\simple-node-server\package.json:

{
  "name": "simple-node-server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Kris Hill",
  "license": "ISC"
}


Is this OK? (yes)
```

Node JS installation

- Step 4: **Start running NPM in your project folder**
  - a file called package.json in your project

```
{
  "name": "u4_bsection",
  "version": "1.0.0",
  "description": "demo",
  "main": "NewU4L2.js",
  ▷ Debug
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
```

Node JS installation

- Step 5: **Install Any NPM Packages. (covered in later classes)**
  - there are two parts to connecting a package to our app:
    - Download/install the package from NPM
    - Save the package name and version number under “Dependencies” in your package.json
- Step 6: **Create an HTML file**
  - Add a file to your project folder with the extension ‘.html’.
- Step 7: **Create a Node/JavaScript file in the project folder**
  - Add a file to your project folder with the extension ‘.js’.

Node JS installation

- Step 8: **Start the Node Server**
  - by opening up a terminal and running the command:
  - *node filenae.js*

- Step 9: **Visit Your (Local) Site!**

Node JS: Examples

- Create a file with .js extension and run it using the command “node filename.js”
  - console.log("Hello World")
  - const name1=“John";
  - console.log('Hello,',name1);

- Non Blocking I/O

```
setTimeout(()=>{
    console.log("Timer Stopped“)},2000);
console.log("Timer Started")
```

Node JS: Examples

- Fetching a file: To include a module, use the require() function with the name of the module:

```
const fs=require('fs')
fs.stat('U4L2.js', (err,stats)=>{
    if (err) throw err;
    console.log('Stats of U4L2.js',JSON.stringify(stats))
})
```

- Renaming a file

```
const fs=require('fs')
fs.rename('U4L2.js',"NewU4L2.js",(err)=>{
    console.log("Rename Succesfull")
})
fs.stat('NewU4L2.js', (err,stats)=>{
    if (err) throw err;
    console.log('Stats of newer.js',JSON.stringify(stats))
})
```

Node JS: Examples

- Non Blocking I/O

```
const fs=require('fs')
fs.readFile('NewU4L2.js','UTF-8' ,(err,data)=>{
    if(err) throw err
    console.log("Contents:",data)
})
console.log("Reading the Contents");
```

- Blocking I/O

```
const fs=require('fs')
const data=fs.readFileSync("NewU4L2.js",'UTF-8')
console.log("Reading the file contents...")
console.log("data:",data)
```

THANK YOU

Revathi G P
Department of
Computer Science and Engineering