Getting started with Node.js

How JavaScript works

- **JavaScript is Single Threaded.**
- **A JavaScript engine exists in a single OS process and consumes a single thread.**
- **When the application is running, CPU execution is never performed in parallel, since the JavaScript engine uses this method, it is impossible for users to get the Deadlocks and Race Conditions which actually makes Multi Threaded applications so complex.**

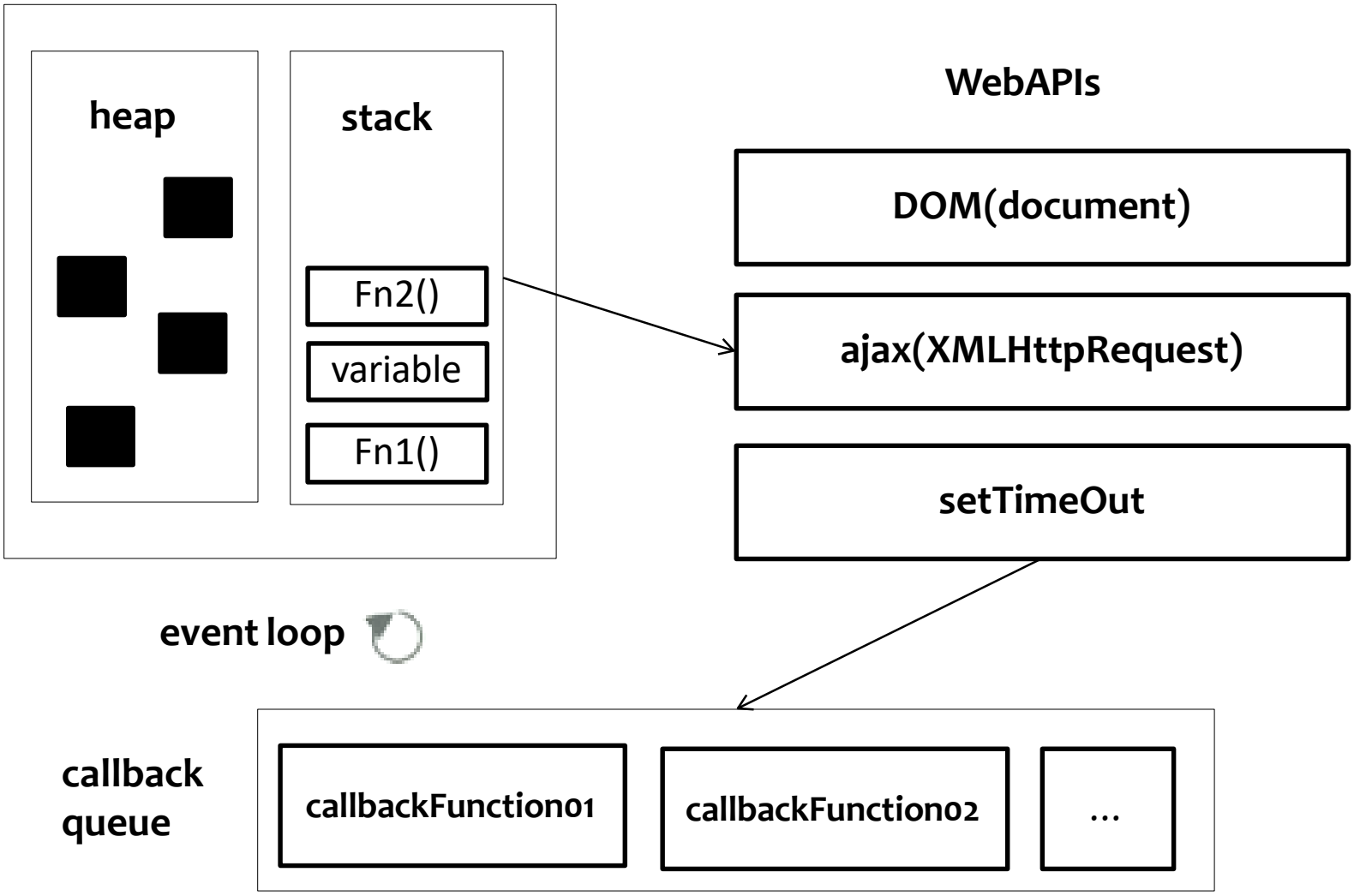
Event loop

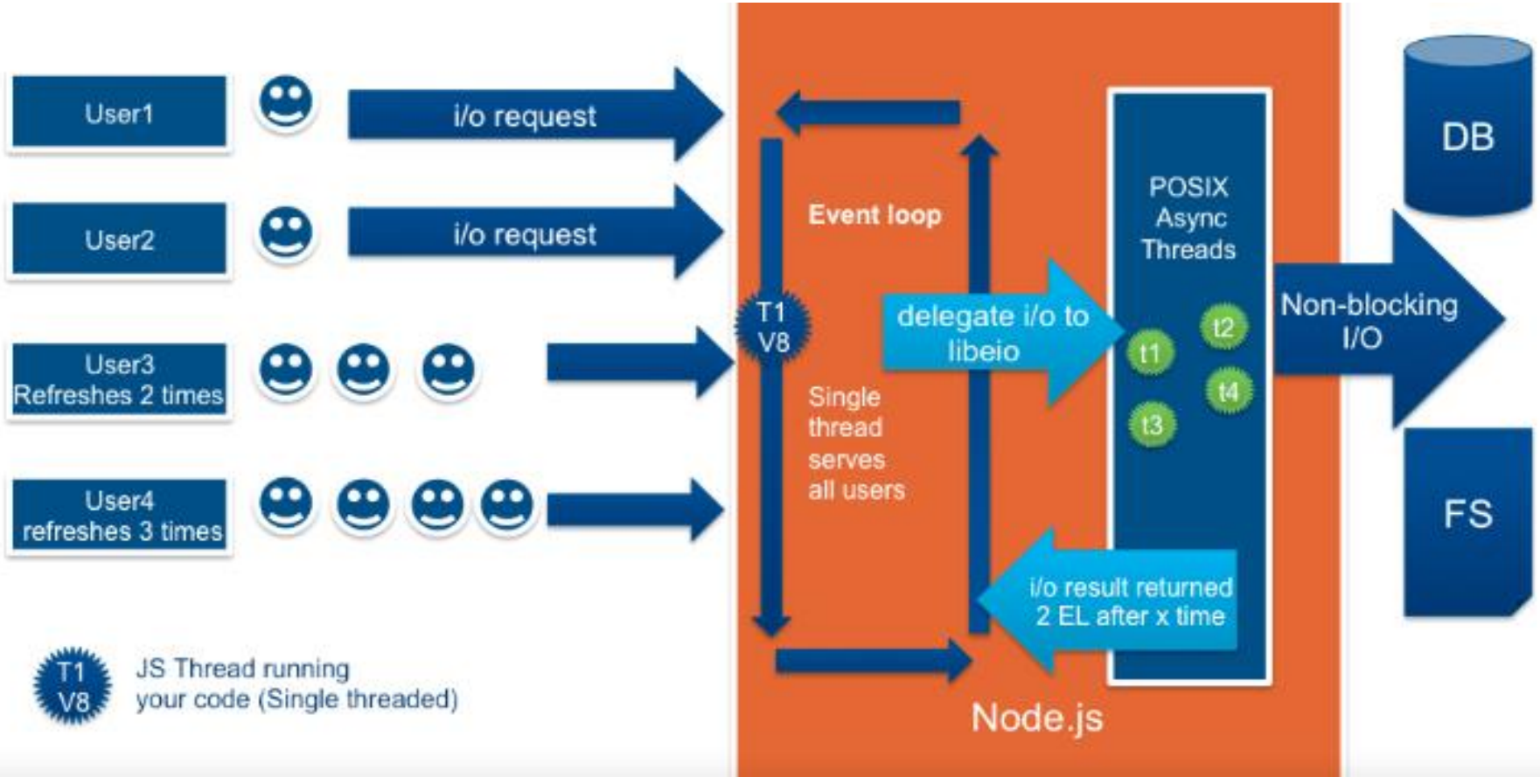
- **An event loop is a construct that mainly performs two functions in a continuous loop**
  - **Event detection** : In any run of the loop, it has to detect which events just happened.
  - **Event handler triggering** : When an event happens, the event loop must determine the event callback and invoke it.
- **Event loop is just one thread running inside one process, which means that, when an event happens, the event handler can run without interruption**
  - There is at most one event handler running at any given time
  - Any event handler will run to completion without being interrupted
- **This allows the programmer to relax the synchronization requirements and not have to worry about concurrent threads of execution changing the shared memory state.**

Stack, Heap and Queue

- **Different browsers have different JavaScript engines (e.g. Chrome has V8, Firefox has OdinMonkey and IE has Chakra)**
- **JavaScript engine has three important features. They are Stack, Heap and Queue**
- **Each browser will implement these features differently, but all does the same.**
- **Stack**
  - Currently running functions gets added to the stack(frame). Pops out from the once it completes its execution.
- **Heap**
  - Memory allocation happened here. It is a bunch of memory where object's live in a unordered manner.
- **Queue**
  - function calls are queued up which gets added to the stack once it is empty.

Stack, Heap and Queue




User1
User2
User3
Refreshes 2 times
User4
refreshes 3 times
i/o request
i/o request
Event loop
T1
V8
Single
thread
serves
all users
delegate i/o to
libeio
POSIX
Async
Threads
t1
t2
t3
t4
Non-blocking
I/O
DB
FS
i/o result returned
2 EL after x time
Node.js
T1
V8
JS Thread running
your code (Single threaded)

- Web-Socket server is created on a single thread — event loop which listens continuously on port 4000.
- When a web or app client connects to it, it fires the ‘onConnection’ event which the loop picks up and immediately publishes to the thread pool and is ready to receive the next request
- This is the main functionality differentiation between NodeJs based servers and other IIS/ Apache based servers, NodeJs for every connection request do not create a new thread instead it receives all request on single thread and delegates it to be handled by many background workers to do the task as required.
- Libuv library handles this workers in collaboration with OS kernel.
- Libuv is the magical library that handles the queueing and processing of asynchronous events utilizing powerful kernel, today most modern kernels are multi-threaded, they can handle multiple operations executing in the background.
- When one of these operations completes, the kernel tells Node.js so that the appropriate callback may be added to the poll queue to eventually be executed.

Introduction to Node.js

- In 2009 Ryan Dahl created Node.js or Node, a framework primarily used to create highly scalable servers for web applications. It is written in C++ and JavaScript.
- Node.js is a platform built on Chrome's JavaScript runtime(v8 JavaScript Engine) for easily building fast, scalable network applications.
- It's a highly scalable system that uses asynchronous, non-blocking I/O model (input/output), rather than threads or separate processes
- It is not a framework like jQuery nor a programming language like C# or JAVA . It's a new kind of web server like has a lot in common with other popular web servers, like Microsoft's Internet Information Services (IIS) or Apache
- IIS / Apache processes only HTTP requests, leaving application logic to be implemented in a language such as PHP or Java or ASP.NET. Node removes a layer of complexity by combining server and application logic in one place.

Why Node.js?

- JavaScript everywhere i.e. Server-side and Client-side applications in JavaScript.
- Node is very easy to set up and configure.
- Vibrant Community
- Small core but large community so far we have 60,000 + packages on npm
- Real-time/ high concurrency apps (I/O bound)
- API tier for single-page apps and rich clients(iOS, Android)
- Service orchestration
- Top corporate sponsors like Microsoft, Joyent, PayPal etc..
- Working with NOSQL(MongoDB) Databases

Traditional Programming Limitations

- **In traditional programming I/O (database, network, file or inter-process communication) is performed in the same way as it does local function calls.**

  i.e. Processing cannot continue until the operation is completed.
- **When the operation like executing a query against database is being executed, the whole process/thread idles, waiting for the response. This is termed as "Blocking"**
- **Due to this blocking behavior we cannot perform another I/O operation, the call stack becomes frozen waiting for the response.**
- **We can overcome this issue by creating more call stacks or by using event callbacks.**

Creating more call stacks

- **To handle more concurrent I/O, we need to have more concurrent call stacks.**
- **Multi-threading is one alternative to this programming model.**
  - Makes use of separate CPU Cores as "Threads"
  - Uses a single process within the Operating System
- **If the application relies heavily on a shared state between threads accessing and modifying shared state increase the complexity of the code and It can be very difficult to configure, understand and debug.**

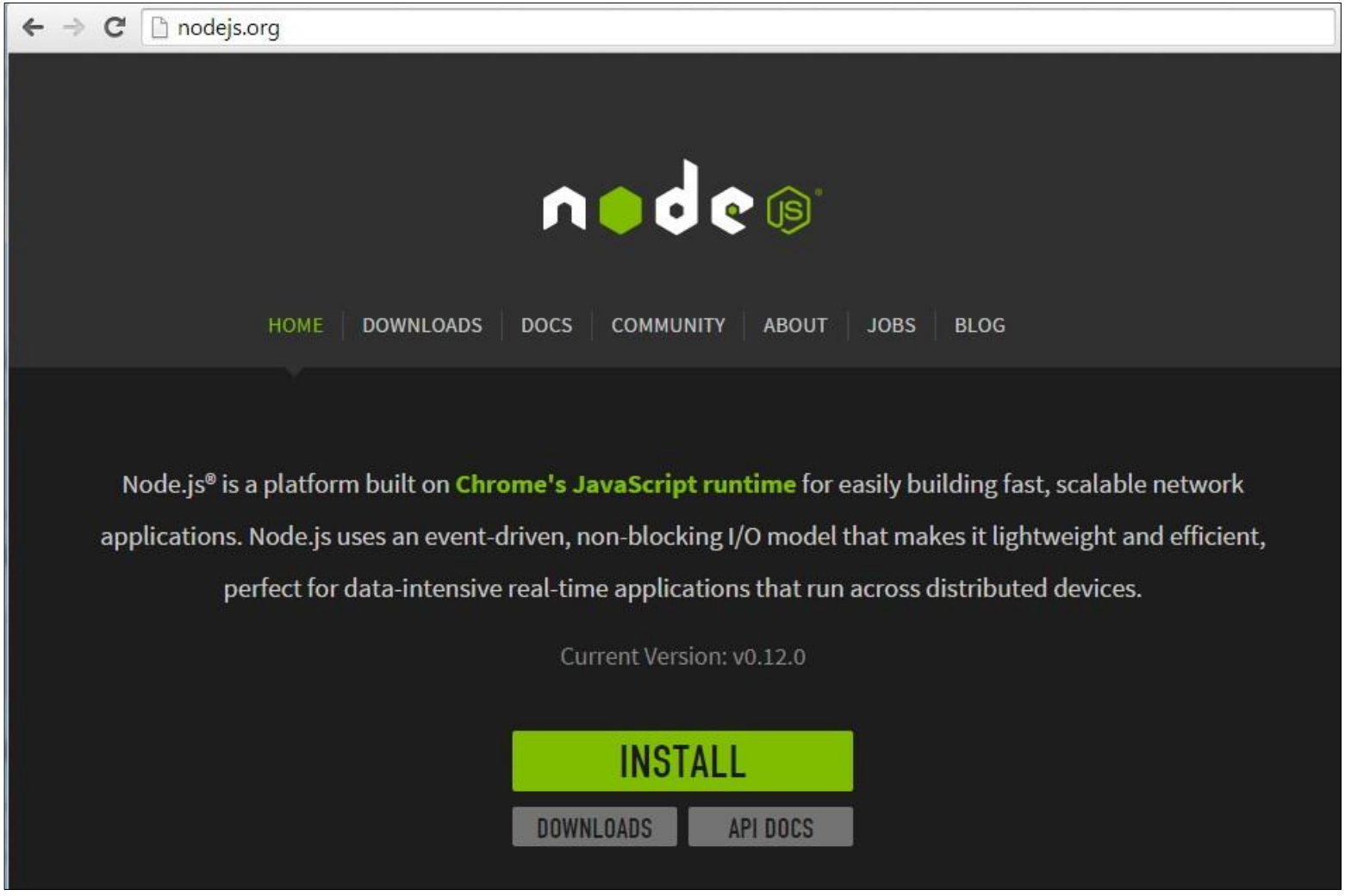
Event-driven Programming

- **Event-driven programming or Asynchronous programming is a programming style where the flow of execution is determined by events.**
- **Events are handled by event handlers or event callbacks**
- **An event callback is a function that is invoked when something significant happens like when the user clicks on a button or when the result of a database query is available.**

```
query_finished = function(result) {
    do_something_with(result);
}
query('SELECT Id,Name FROM employees', query_finished);
```

- **Now instead of simply returning the result, the query will invoke the query_finished function once it is completed.**
- **Node.js supports Event-driven programming and all I/O in Node are non-blocking**

nodejs.org – Node.js official Website



Downloading Node.js

Windows Installer

node-v0.12.0-x86.msi

Macintosh Installer

node-v0.12.0.pkg

Source Code

node-v0.12.0.tar.gz

Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.exe)	32-bit	64-bit
Mac OS X Installer (.pkg)	Universal	
Mac OS X Binaries (.tar.gz)	32-bit	64-bit
Linux Binaries (.tar.gz)	32-bit	64-bit
SunOS Binaries (.tar.gz)	32-bit	64-bit
Source Code	node-v0.12.0.tar.gz	

Node.js Globals

- **global**
  - Any variables or members attached to global are available anywhere in our application. GLOBAL is an alias for global
    - global.companyName = 'KLFS'
    - global['companyName'] // We can directly access the members attached to global.
- **process**
  - The process object is a global object which is used to Inquire the current process to know the PID, environment variables, platform, memory usage etc.
    - process.platform
    - process.exit( )
- **console**
  - It provides two primary functions for developers testing web pages and applications
    - console.log('KLFS')

Module Introduction

- **A module is the overall container which is used to structure and organize code.**
- **It supports private data and we can explicitly defined public methods and variables (by just adding/removing the properties in return statement) which lead to increased readability.**
- **JavaScript doesn't have special syntax for package / namespace, using module we can create self-contained decoupled pieces of code.**
- **It avoids collision of the methods/variables with other global APIs.**

Modules in Node.js

- **In Node, modules are referenced either by file path or by name.**
- **Node's core modules expose some Node core functions(like global, require, module, process, console) to the programmer, and they are preloaded when a Node process starts.**
- **To use a module of any type, we have to use the require function. The require function returns an object that represents the JavaScript API exposed by the module.**
  - *var module = require('module_name');*

Loading a module

- **Modules can be referenced depending on which kind of module it is.**
- **Loading a core module**
  - Node has several modules compiled into its binary distribution. These are called the core modules. It is referred solely by the module name, not by the path and are preferentially loaded even if a third-party module exists with the same name.
    - *var http = require('http');*
- **Loading a file module (User defined module)**
  - We can load non-core modules by providing the absolute path / relative path. Node will automatically adding the .js extension to the module referred.
    - *var myModule = require('d:/Rojrocks/nodejs/module'); // Absolute path for module.js*
    - *var myModule = require('../module'); // Relative path for module.js (one folder up level)*
    - *var myModule = require('./module'); // Relative path for module.js (Exists in current directory)*

Loading a module

- **Loading a folder module (User defined module)**
  - We can use the path for a folder to load a module.
    - *var myModule = require('./myModuleDir');*
  - Node will presume the given folder as a package and look for a package definition file inside the folder. Package definition file name should be named as ***pagkage.json***
  - Node will try to parse ***package.json*** and look for and use the ***main*** attribute as a relative path for the entry point.
  - We need to use ***npm init*** command to create ***package.json***.

- **Creating Package.json using npm init command**

D:\Rojrocks\mynodejsApp> npm init

{ "name": "Rojrocks_Modules",
"version": "1.0.0",
"description": "Rojrocks Modules for Demo",
"main": "index.js",
"scripts": { "test": "echo \"Error: no test specified\" && exit 1" },
"author": "Rojrocks M <Rojrocks.khadilkar@KLFS.com>",
"dependencies":{
}
"devdependencies:{

}
"license": "ISC"}

- ***var myModule = require('d:/Rojrocks/ mynodejsApp'); // refer index.js placed in modules folder***

package.json usage

- **package.json is a configuration file from where the npm can recognize dependencies between packages and installs modules accordingly.**
- **It must be located in project's root directory.**
- **The JSON data in package.json is expected to adhere to a certain schema. The following fields are used to build the schema for package.json file**
  - **name and version** : package.json must be specified at least with a name and version for package. Without these fields, npm cannot process the package.
  - **description and keywords** : description field is used to provide a textual description of package. Keywords field is used to provide an array of keywords to further describe the package. It is used by npm search command
  - **author** : The primary author of a project is specified in the author field.
  - **main** : Instruct Node to identify its main entry point.

package.json usage

- **dependencies** : Package dependencies are specified in the dependencies field.
- **devdependencies** : Many packages have dependencies that are used only for testing and development. These packages should not be included in the dependencies field. Instead, place them in the separate devdependencies field.
- **scripts :** The scripts field, when present, contains a mapping of npm commands to script commands. The script commands, which can be any executable commands, are run in an external shell process. Two of the most common commands are start and test. The start command launches your application, and test runs one or more of your application's test scripts.

Node Package Manager

- **Loading a module(Third party) installed via NPM (Node Package Manager)**
  - Apart from writing our own modules and core modules, we will frequently use the modules written by other people in the Node community and published on the Internet (npmjs.com).
  - We can install those third party modules using the ***Node Package Manager*** which is installed by default with the node installation.
  - To install modules via npm use the **npm install** command.
  - npm installs module packages to the **node_modules** folder.
  - To update an installed package to a newer version use the **npm update** command.
  - If the module name is not relative and is not a core module, Node will try to find it inside the node_modules folder in the current directory.
    - *var jade = require('jade');*
    - *Here jade is not a core module and not available as a user defined module found in relative path, it will look into the node_modules/jade/package.json and refer the file/ folder mentioned in main attribute.*

Creating and exporting a module

- **Creating a module that exposes / exports a function called helloWorld**

```
// Save it as myModule.js
exports.helloWorld = function () {
    console.log("Hello World");
}
```

- ***exports* object is a special object created by the Node module system which is returned as the value of the require function when you include that module.**

- **Consuming the function on the exports object created in myModule.js**

```
// Save it as moduleTest.js
var module = require('./myModule');
module.helloWorld();
```

- **We can replace exports with module.exports**
  - exports = module.exports = { }

Creating and exporting a module

```
var myVeryLongInternalName = function() { ... };
exports.shortName = myVeryLongInternalName;
// add other objects, functions, as required
followed by:
```

```
var m = require('./mymodule');
m.shortName(); // invokes module.myVeryLongInternalName
```

Buffers in Node

- JavaScript doesn't have a byte type. It just has strings.
- Node is based on JavaScript with just using string type it is very difficult to perform the operations like communicate with HTTP protocol, working with databases, manipulate images and handle file uploads.
- Node includes a binary buffer implementation, which is exposed as a JavaScript API under the Buffer pseudo-class.
- Using buffers we can manipulate, encode, and decode binary data in Node. In node each buffer corresponds to some raw memory allocated outside V8.
- A buffer acts like an array of integers, but cannot be resized

Creating Buffers in Node

- **new Buffer(n) is used to create a new buffer of 'n' octets. One octet can be used to represent decimal values ranging from 0 to 255.**
- **There are several ways to create new buffers.**
  - **new Buffer(n)** : To create a new buffer of 'n' octets
    - var buffer = new Buffer(10);
  - **new Buffer(arr)** : To create a new buffer, using an array of octets.
    - var buffer = new Buffer([7,1,4,7,0,9]);
  - **new Buffer(str,[encoding])** : To create a new buffer, using string and encoding.
    - var buffer = new Buffer("KLFS","utf-8"); // utf-8 is the default encoding in Node.

Writing to Buffer

- **Writing to Buffer**
  - buf.write(str, [offset], [length], [encoding]) method is used to write a string to the buffer.
  - buf.write() 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.
  - An offset or the index of the buffer to start writing at. Default value is 0.

Reading from Buffer

- **Reading from buffers**
  - buf.toString([encoding], [start], [end]) method decodes and returns a string from buffer data.
  - buf.toString() returns method reads the entire buffer and returns as a string.
  - buf.toJSON() method is used to get the JSON-representation of the Buffer instance, which is identical to the output for JSON Arrays.

Event Handling in Node

- **In node there are two event handling techniques. They are called callbacks and EventEmitter.**
- **Callbacks are for the async equivalent of a function. Any async function in node accepts a callback as it's last parameter**

```
var myCallback = function(data) {
  console.log('got data: '+data);
};

var fn = function(callback) {
    callback('Data from Callback');
};

fn(myCallback);
```

EventEmitter

- **In node.js an event can be described simply as a string with a corresponding callback and it can be emitted.**
- **The *on* or *addListener* method allows us to subscribe the callback to the event.**
- **The emit method "emits" event, which causes the callbacks registered to the event to trigger.**

```
var events = require('events');
var eventEmitter = new events.EventEmitter();
var myCallback = function(data) {
    console.log('Got data: '+data);
};

eventEmitter.on('RojrocksEvent', myCallback);
var fn = function() {
     eventEmitter.emit('RojrocksEvent','Data from Emitter');
};
fn();
```

EventEmitter Methods

- **All objects which emit events in node are instances of events.EventEmitter which is available inside Event module.**
- **We can access the Event module using require("events")**
- **addListener(event, listener) / on(event, listener)**
  - Adds a listener to the end of the listeners array for the specified event. Where listener is a function which needs to be executed when an event is emitted.
- **once(event, listener)**
  - Adds a one time listener for the event. This listener is invoked only the next time the event is fired, after which it is removed.
- **removeListener(event, listener)**
  - Remove a listener from the listener array for the specified event
- **removeAllListeners([event])**
  - Removes all listeners, or those of the specified event

Creating an EventEmitter

- **We can create Event Emitter and customevents**

```
var EventEmitter = require('events').

Var eventEmmiter=new events.EventEmitter();

//event handler function

var mycallback =  function(data){

   console.log(data);

 }
//configure the event
eventEmitter.on('RojRocksEvent',mycallback);

//emit will generate event
Var fn = function() {  evenEmitter.emit('RojRocksEvent', 'Data
     from fn function');
}


fn();
```

File System Module

- By default Node.js installations come with the file system module.
- This module provides a wrapper for the standard file I/O operations.
- We can access the file system module using

```
require("fs")
```

File System Module

- All the methods in this module has
  - asynchronous forms
  - synchronous forms.
- synchronous methods in this module ends with 'Sync'. For instance *renameSync* is the synchronous method for *rename* asynchronous method.
- The asynchronous form always take a completion callback as its last argument.
  - The arguments passed to the completion callback depend on the method,
  - but the first argument is always reserved for an exception.
  - If the operation was completed successfully, then the first argument will be null or undefined.
- When using the synchronous form any exceptions are immediately thrown. You can use try/catch to handle exceptions or allow them to bubble up.

File I/O methods

- **fs.stat(path, callback)**
  - Used to retrieve meta-info on a file or directory.
- **fs.readFile(filename, [options], callback)**
  - Asynchronously reads the entire contents of a file.
- **fs.writeFile(filename, data, [options], callback)**
  - Asynchronously writes data to a file, replacing the file if it already exists. Data can be a string or a buffer.
- **fs.unlink(path, callback)**
  - Asynchronously deletes a file.
- **fs.watchFile(filename, [options], listener)**
  - Watch for changes on filename. The callback listener will be called each time the file is accessed. Second argument is optional by default it is { persistent: true, interval: 5007 }. The listener gets two arguments the current stat object and the previous stat object.

File I/O methods

- **fs.exists(path, callback)**
  - Test whether or not the given path exists by checking with the file system. The callback argument assigned with either true or false based on the existence.
- **fs.rmdir(path, callback)**
  - Asynchronously removes the directory.
- **fs.mkdir(path, [mode], callback)**
  - Asynchronously created the directory.
- **fs.open(path, flags, [mode], callback)**
  - Asynchronously open the file.
- **fs.close(fd, callback)**
  - Asynchronously closes the file.
- **fs.read(fd, buffer, offset, length, position, callback)**
  - Read data from the file specified by fd.

Asynchronous read file

```
var fs=require('fs');
fs.readFile('./demo.txt', "utf-8", function (err, data) {
        if (err) console.log("error");
        console.log(data);
});

console.log("Program ends here");
console.log("Program ends here123");
```

Synchronous read

```
var fs=require('fs');
var content = fs.readFileSync('./demo.txt', "utf-8");
console.log(content);
console.log("program ends here");
```

```
var fs = require("fs");
var path = __dirname + "/sample.txt";
fs.stat(path, function(error, stats) {
            //open file in read mode
             fs.open(path, "r", function(error, fd) {
                        //create  buffer of file size
                        var buffer = new Buffer(stats.size);
                         //read file and store contents in buffer
                        fs.read(fd, buffer, 0, buffer.length, null,
                                  function(error, bytesRead, buffer) {
                                             var data = buffer.toString("utf8");
                                             console.log(data);
                                                        });
                        });
});
console.log('program ends here');
```

Read file line by line

- Node.js provides a built-in module readline that can read data from a readable stream.
- It emits an event- “line” whenever the data stream encounters an end of line character (\n, \r, or \r\n).
- **Readline Module**

The readline module is inbuilt into Node, so you don't need to install any third party module

- A special thing to note is that the readline module reads from a stream rather than bufferring the whole file in memory (read Node.js Streams). That is why the createReadStream method is used.

```
var fs = require("fs");
var path = __dirname + "/sampleout.txt";
var data = "This data will be written in the file";
//open file it will assign fd
fs.open(path, "w", function(error, fd) {
    //data will be stored in the buffer
    var buffer = new Buffer(data);
    //buffer contents will be written in the file
    fs.write(fd, buffer, 0, buffer.length, null,
                        function(error, written, buffer) {
            if (error) {
                console.error("write error: " + error.message);
             } else {
                  console.log("Successfully wrote "   + written + " bytes.");
              }
          });
});
```

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

// create instance of readline
// each instance is associated with single input stream
let rl = readline.createInterface({
   input: fs.createReadStream('products.txt')
});

let line_no = 0;

// event is emitted after each line
rl.on('line', function(line) {
   line_no++;
   console.log(line);
});

// end
rl.on('close', function(line) {
   console.log('Total lines : ' + line_no);
});
```

Stream

- A stream is an abstract interface implemented by various objects in Node. They represent inbound (ReadStream) or outbound (WriteStream) flow of data.
- Streams are readable, writable, or both (Duplex).
- All streams are instances of EventEmitter.
- Stream base classes can be loaded using *require('stream')*
- *ReadStream* is like an outlet of data, once it is created we can wait for the data, pause it, resume it and indicates when it is actually end.
- *WriteStream* is an abstraction on where we can send data to. It can be a file or a network connection or even an object that outputs data that was transformed(when zipping a file)

Readable Stream

- ***ReadStream* is like an outlet of data, which is an abstraction for a source of data that you are reading from**
- **A Readable stream will not start emitting data until you indicate that you are ready to receive it.**
- **Readable streams have two "modes": a flowing mode and a non-flowing mode.**
  - In flowing mode, data is read from the underlying system and provided to your program as fast as possible.
  - In non-flowing mode, you must explicitly call stream.read() to get chunks of data out.
- **Readable streams can emit the following events**
  - **'readable'** : This event is fired when a chunk of data can be read from the stream.
  - **'data'** : This event is fired when the data is available. It will switch the stream to flowing mode when it is attached. It is the best way to get the data from stream as soon as possible.

Readable Stream

- **'end'** : This event is fired when there will be no more data to read.
- **'close'** : Emitted when the underlying resource (for example, the backing file descriptor) has been closed. Not all streams will emit this.
- **'error'** : Emitted if there was an error receiving data.

➢ **Readable streams has the following methods**

- **readable.read([size])** : Pulls data out of the internal buffer and returns it. If there is no data available, then it will return null.
- **readable.setEncoding(encoding)** : Sets the encoding to use.
- **readable.resume()** : This method will cause the readable stream to resume emitting data events.
- **readable.pause()** : This method will cause a stream in flowing-mode to stop emitting data events. Any data that becomes available will remain in the internal buffer.
- **readable.pipe(destination, [options])** : Pulls all the data out of a readable stream and writes it to the supplied destination, automatically managing the flow.

Writable Stream

- **Writable stream interface is an abstraction for a destination that you are writing data to.**
- **Writable streams has the following methods**
  - **writable.write(chunk, [encoding], [callback])** : This method writes some data to the underlying system and calls the supplied callback once the data has been fully handled. Here chunk is a String / Buffer data to write
  - **writable.end([chunk], [encoding], [callback])** : Call this method when no more data will be written to the stream. Here chunk String / Buffer optional data to write
- **Writable streams can emit the following events**
  - **'drain'** : If a writable.write(chunk) call returns false, then the drain event will indicate when it is appropriate to begin writing more data to the stream.
  - **'finish'** : When the end() method has been called, and all data has been flushed to the underlying system, this event is emitted

Writable Stream

- **'pipe' :** This is emitted whenever the pipe() method is called on a readable stream, adding this writable to its set of destinations.
- **'unpipe' :** This is emitted whenever the unpipe() method is called on a readable stream, removing this writable from its set of destinations.
- **'error' :** Emitted if there was an error when writing or piping data.

Creating a big file

```
const fs = require('fs');
const file = fs.createWriteStream('./big.file');

for(let i=0; i<= 1e6; i++) {
  file.write('Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.\n');
}

file.end();
```

Inefficient way of reading big file

```
const fs = require('fs');
const server = require('http').createServer();

server.on('request', (req, res) => {
  fs.readFile('./big.file', (err, data) => {
    if (err) throw err;

    res.end(data);
  });
});

server.listen(8000);
```

This will store the data in buffer i.e in the memory, so cannot read files with size greater than RAM size
Better to use pipe command

readableSrc.pipe(writableDest)
When we ran the server, it started out with a normal amount of memory, 8.7 MB:

the memory consumption jumped to 434.8 MB.

Efficient way of reading files

```
const fs = require('fs');
const server = require('http').createServer();

server.on('request', (req, res) => {
  const src = fs.createReadStream('./big.file');
  src.pipe(res);
});

server.listen(8000);
```

When a client asks for that big file, we stream it one chunk at a time, which means we don’t buffer it in memory at all. The memory usage grew by about 25 MB and that’s it.

Introduction

- **Webserver like IIS / Apache serves static files(like html files) so that a browser can view them over the network.**
- **We need to place the files in a proper directory(like wwwroot in IIS), so that we can navigate to it using http protocol. The web server simply knows where the file is on the computer and serves it to the browser.**
- **Node offers a different paradigm than that of a traditional web server i.e. it simply provides the framework to build a web server.**
- **Interestingly building a webserver in node is not a cumbersome process, it can be written in just a few lines, moreover we'll have full control over the application.**

2 HTTP module in Node.js

- We can easily create an HTTP server in Node.
- To use the HTTP server and client one must require('http').
- The HTTP interfaces in Node are designed to support many features of the protocol which have been traditionally difficult to use. In particular, large, possibly chunk-encoded, messages.
- Node's HTTP API is very low-level. It deals with stream handling and message parsing only. It parses a message into headers and body
- HTTP response implements the Writable Stream interface and request implements Readable Stream interface.

Creating HTTP Server

```
/* Loading http module*/
var http = require('http');

/* Returns a new web server object*/
var server = http.createServer(function(req,res){

        /* Sends a response header to the request.*/
        res.writeHead(200,{'content-type':'text/html'});

        /*sends a chunk of the response body*/
        res.write('<h1>Hello KLFS</h1>')

         /* signals server that all the responses has been sent */
        res.end('<b>Response Ended</b>')

});

/* Accepting connections on the specified port and hostname. */
server.listen(3000);

console.log('server listening on localhost:3000');
```

Routing

- **Routing refers to the mechanism for serving the client the content it has asked**

```
var http = require('http');
var server = http.createServer(function(req,res){
var path = req.url.replace(/\/?(?:\?.*)?$/, '').toLowerCase();
switch(path) {
        case '' :
                res.writeHead(200, {'Content-Type': 'text/html'});
                res.end('<h1>Home Page</h1>');
                break;
        case '/about' :
                res.writeHead(200, {'Content-Type': 'text/html'});
                res.end('<h1>About us</h1>');
                break;
        default:
                res.writeHead(404, { 'Content-Type': 'text/plain' });
                res.end('Not Found');
                break;
        }
});
server.listen(3000);
```

Accept data through form form.html

```
<!doctype html>
    <html>
    <body>
      <form action="/calc" method="post">
        <input type="text" name="num1" /><br />
        <input type="number" name="num2" /><br />

        <button type="submit">Save</button>
      </form>
    </body>
    </html>
```

Accept data from user through form

```
var http=require("http");
var fs=require("fs");
var url=require("url");
var query=require("querystring");
//add user defined module
var m1=require("./formaddmodule");
//function to handle request and response
function process_request(req,resp)
{
//to parse url to separate path
   var u=url.parse(req.url);
   console.log(u);
//response header
   resp.writeHead(200,{'Content-
Type':'text/html'});
```

Continued

```
//routing info
switch(u.pathname){
   case '/':
            // read data from html file
            fs.readFile("form.html",function(err,data){
              if(err){
                        resp.write('some error');
                        console.log(err);

              }else{
              // write file data to response
              resp.write(data);
              //send response to client
              resp.end();}
   });
break;
```

```
case '/calc':
                var str="";
                //data event handling while receiving data
                req.on('data',function(d){
                str+=d;});
                //end event when finished receiving data
                req.on('end',function(){
                  console.log(str);
                            var ob=query.parse(str);
                            //calling user defined function from user module
                            var sum=m1.add(ob.num1,ob.num2);
                            resp.end("<h1>Addition : "+sum+"</h1>");

                });

 }

}
var server=http.createServer(process_request);
server.listen(3000);
```

Forms add module

Formaddmodule.js

```
exports.add=function(a,b){

   return parseInt(a)+parseInt(b);

}
```

Introduction

- **If we try to create apps by only using core Node.js modules we will end up by writing the same code repeatedly for similar tasks such as**
  - Parsing of HTTP request bodies
  - Parsing of cookies
  - Managing sessions
  - Organizing routes with a chain of if conditions based on URL paths and HTTP methods of the requests
  - Determining proper response headers based on data types
- **Developers have to do a lot of manual work themselves, such as interpreting HTTP methods and URLs into routes, and parsing input and output data.**
- **Express.js solves these and many other problems using abstraction and code organization.**

Working with Express framework

Introduction toExpress.js

- Express.js is a web framework based on the core Node.js http module and Connect components
- Express.js framework provides a model-view-controller-like structure for your web apps with a clear separation of concerns (views, routes, models)
- Express.js systems are highly configurable, which allows developers to pick freely whatever libraries they need for a particular project
- Express.js framework leads to flexibility and high customization in the development of web applications.
- In Express.js we can define middleware such as error handlers, static files folder, cookies, and other parsers.
- Middleware is a way to organize and reuse code, and, essentially, it is nothing more than a function with three parameters: request, response, and next.

Connect module

- **Connect is a module built to support interception of requests in a modular approach.**

```
var logger = function(req, res, next) {
    console.log(req.method, req.url);
    next();
};
var helloWorld = function(req, res, next) {
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello World');
};
app.use(logger);
app.use('/hello',helloWorld);
app.listen(3000);
console.log('Server running at localhost:3000');
```

Express.js Installation

- **The Express.js package comes in two flavors:**
- ***express-generator*: a global NPM package that provides the command-line tool for rapid app creation (scaffolding)**
- ***express*: a local package module in your Node.js app's node_modules folder**

Package.json

```
{  "name": "myapp",
 "version": "1.0.0",
  "description": "This is description",
"main": "index.js",
 "scripts": {    "test": "echo \"Error: no test specified\" && exit 1"  },
"dependencies":{      "express":"",      "body-parser":""  },
  "devDependencies":{    },
"keywords": [    "asd",    "sdjl",    "ljdfljsd",    "kjdflkjlkj"  ],
 "author": "Kishori",
"license": "ISC"}
```

To install if package.json file in myapp folder

C:/…../myapp>npm install

app.js

- **App.js is the main file in Express framework. A typical structure of the main Express.js file consists of the following areas**
  - 1. Require dependencies
  - 2. Configure settings
  - 3. Connect to database (optional)
  - 4. Define middleware
  - 5. Define routes
  - 6. Start the server

- **The order here is important, because requests travel from top to bottom in the chain of middleware.**

app.js

- **Routes are processed in the order they are defined. Usually, routes are put after middleware, but some middleware may be placed following the routes. A good example of such middleware, found after a routes, is error handler.**
- **The way routes are defined in Express.js is with helpers app.VERB(url, fn1, fn2, ..., fn), where fnNs are request handlers, url is on a URL pattern in RegExp, and VERB values are as follows:**
  - all: catch every request (all methods)
  - get: catch GET requests
  - post: catch POST requests
  - put: catch PUT requests
  - del: catch DELETE requests

app.js

- **Finally to start the server, using express object**

```
var express=require("express");
var app=express();
 app.listen(3000, function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Handling post request

POST request

```
var express=require("express");
var app=express();
var bodyparser=require("body-parser");

app.use(bodyparser.urlencoded({extended:false}));
app.use(function(req,resp,next){
                console.log(req.method+"------"+req.url);
                next();
});
app.get("/",function(req,resp){
  resp.sendFile("form.html",{root:__dirname});
});
app.post("/calc",function(req,resp){
              resp.send("<h1>"+req.body.num1+"------"+req.body.num2+"</h1>");
});



app.listen(2000);
```

```
//import the express module
var express = require('express');

//store the express in a variable
var app = express();

//allow express to access our html (index.html) file
app.get('/index.html', function(req, res) {
      res.sendFile(__dirname + "/" + "index.html");
   });
```

Handling GET request in express

Handling Get Request

```
app.get('/user', function(req, res){
        response = {
            first_name : req.query.first_name,
            last_name : req.query.last_name,
            gender: req.query.gender
            };
```

User-info.html

```
<html>
 <head>
  <title>Basic User Information</title>
 </head>
 <body>
    <form action="http://localhost:8888/process_get" method="GET">
        First Name: <input type="text" name="first_name"> <br>
        Last Name: <input type="text" name="last_name"> <br>
          Gender: <select name="gender">
                  <option value="Male">Male</option>
                  <option value="Female">Female</option>
                 </select>
            <input type="submit" value="submit">
    </form>
 </body>
</html>
```

```
//route the GET request to the specified path, "/user".
//This sends the user information to the path
app.get('/user', function(req, res){
      response = {
         first_name : req.query.first_name,
         last_name : req.query.last_name,
         gender: req.query.gender
         };

      //this line is optional and will print the response on the command prompt
      //It's useful so that we know what infomration is being transferred
      //using the server
      console.log(response);

      //convert the response in JSON format
      res.end(JSON.stringify(response));
   });
```

```
//This piece of code creates the server
//and listens to the request at port 8888
//we are also generating a message once the
//server is created
var server = app.listen(8888, function(){
      var host = server.address().address;
      var port = server.address().port;
      console.log("Example app listening at http://%s:%s", host, port);
   });
```

Steps for creating Express.js Application

- **Step – 1 : Create a folder named MyApp**
- **Step – 2 : Create package.json with the following schema**

```
{
  "name": "MyApp",
  "version": "1.0.0",
  "description": "My first express js Application",
  "main": "app.js",
  "dependencies": {
    "body-parser": "1.10.1",
    "cookie-parser": "1.3.3",
    "express": "4.10.7",
    "jade": "1.8.2"
  },
   "scripts": {
    "start": "node app"
  },
  "author": "Kishori",
  "license": "ISC"
}
```

Steps for creating Express.js Application

- **Step – 3 : Install the dependencies using npm install command**
  - D:\Rojrocks\NodeJS\Lesson02\SampleApp>npm install
- **Step – 4 : Create the following folders under MyApp folder**
  - **public** : All the static (front-end) files like HTML
  - **public/css** : Stylesheet files
  - **public/images :** images
  - **public/scripts** : Scripts
  - **db** : Seed data and scripts for MongoDB
  - **views** : Jade (or any other template engine) files
  - **views/includes** : Partial / include files
  - **routes :** Node.js modules that contain request handlers
- **Step – 5 : Create the main file named app.js**

Steps for creating Express.js Application

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

var app = express();

app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

app.all('*', function(req, res) {
    res.render('index',{title: 'KLFS services expressJs training'});
});

http.createServer(app).listen(app.get('port'),function(){
    console.log('Express.js server listening on port ' +app.get('port'));
});
```

Steps for creating Express.js Application

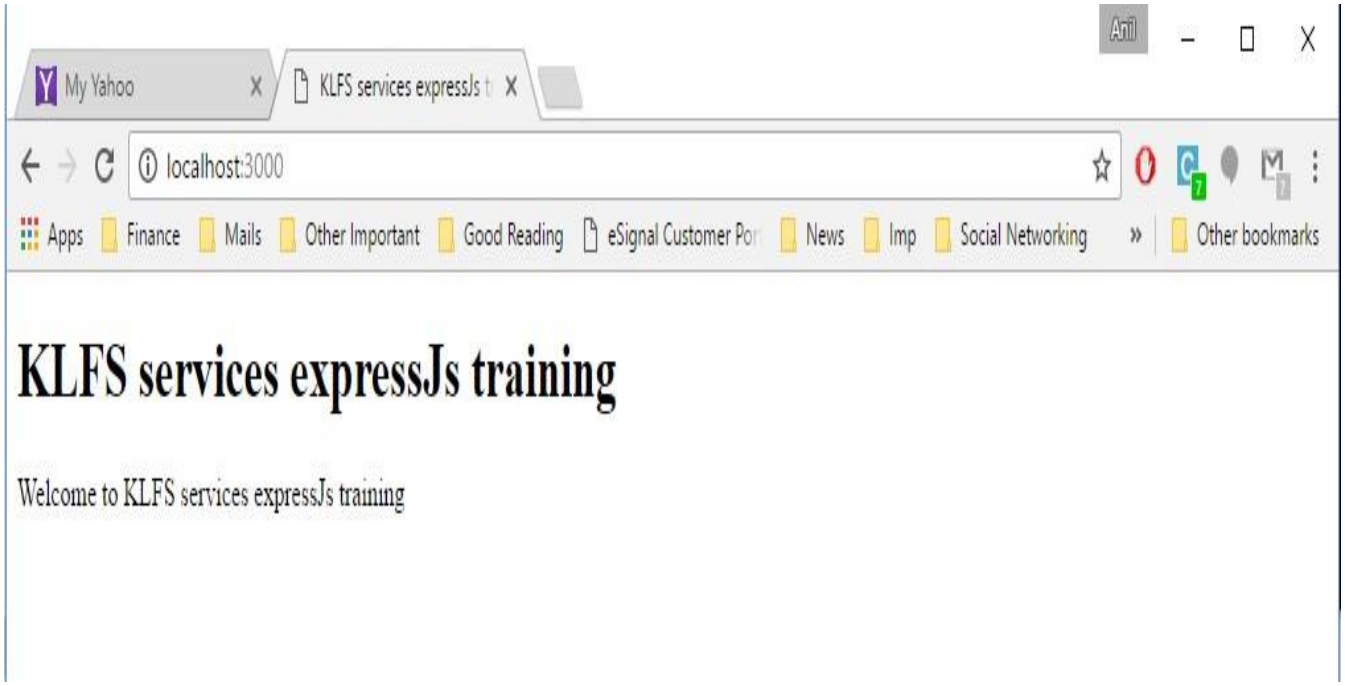
- **Step – 7 : Create index.jade under views folder and type the following contents**

```
doctype html
html
    head
        title= title
    body
        h1= title
        p Welcome to #{title}
```

- **Step – 8 : Start the app by typing *npm start* in command prompt.**

Steps for creating Express.js Application

- **Step – 9 : Open browser and type *http://localhost:3000* to view the SampleApp**



application object properties & methods

Property/Method	Description
`app.set(name, value)`	Sets app-specific properties
`app.get(name)`	Retrieves value set by `app.set()`
`app.enable(name)`	Enables a setting in the app
`app.disable(name)`	Disables a setting in the app
`app.enabled(name)`	Checks if a setting is enabled
`app.disabled(name)`	Checks if a setting is disabled
`app.configure([env], callback)`	Sets app settings conditionally based on the development environment
`app.use([path], function)`	Loads a middleware in the app
`app.engine(ext, callback)`	Registers a template engine for the app
`app.param([name], callback)`	Adds logic to route parameters
`app.VERB(path, [callback...], callback)`	Defines routes and handlers based on HTTP verbs
`app.all(path, [callback...], callback)`	Defines routes and handlers for all HTTP verbs
`app.locals`	The object to store variables accessible from any view
`app.render(view, [options], callback)`	Renders view from the app
`app.routes`	A list of routes defined in the app
`app.listen()`	Binds and listen for connections

request object properties & methods

Property/Method	Description
`req.params`	Holds the values of named routes parameters
`req.params(name)`	Returns the value of a parameter from named routes or `GET` params or `POST` params
`req.query`	Holds the values of a `GET` form submission
`req.body`	Holds the values of a `POST` form submission
`req.files`	Holds the files uploaded via a form
`req.route`	Provides details about the current matched route
`req.cookies`	Cookie values
`req.signedCookies`	Signed cookie values
`req.get(header)`	Gets the request HTTP header
`req.accepts(types)`	Checks if the client accepts the media types
`req.accepted`	A list of accepted media types by the client
`req.is(type)`	Checks if the incoming request is of the particular media type

request object properties & methods

Property/Method	Description
`req.ip`	The IP address of the client
`req.ips`	The IP address of the client, along with that of the proxies it is connected through
`req.stale`	Checks if the request is stale
`req.xhr`	Checks if the request came via an AJAX request
`req.protocol`	The protocol used for making the request
`req.secure`	Checks if it is a secure connection
`req.subdomains`	Subdomains of the host domain name
`req.url`	The request path, along with any query parameters
`req.originalUrl`	Used as a backup for `req.url`
`req.acceptedLanguages`	A list of accepted languages by the client
`req.acceptsLanguage(langauge)`	Checks if the client accepts the language
`req.acceptedCharsets`	A list of accepted charsets by the client
`req.acceptsCharsets(charset)`	Checks if the client accepts the charset
`req.host`	Hostname from the HTTP header

response object properties & methods

Property/Method	Description
`res.status(code)`	Sets the HTTP response code
`res.set(field, [value])`	Sets response HTTP headers
`res.get(header)`	Gets the response HTTP header
`res.cookie(name, value, [options])`	Sets cookie on the client
`res.clearCookie(name, [options])`	Deletes cookie on the client
`res.redirect([status], url)`	Redirects the client to a URL, with an optional HTTP status code
`res.location`	The location value of the response HTTP header
`res.charset`	The charset value of the response HTTP header
`res.send([body|status], [body])`	Sends an HTTP response object, with an optional HTTP response code
`res.json([status|body], [body])`	Sends a JSON object for HTTP response, along with an optional HTTP response code

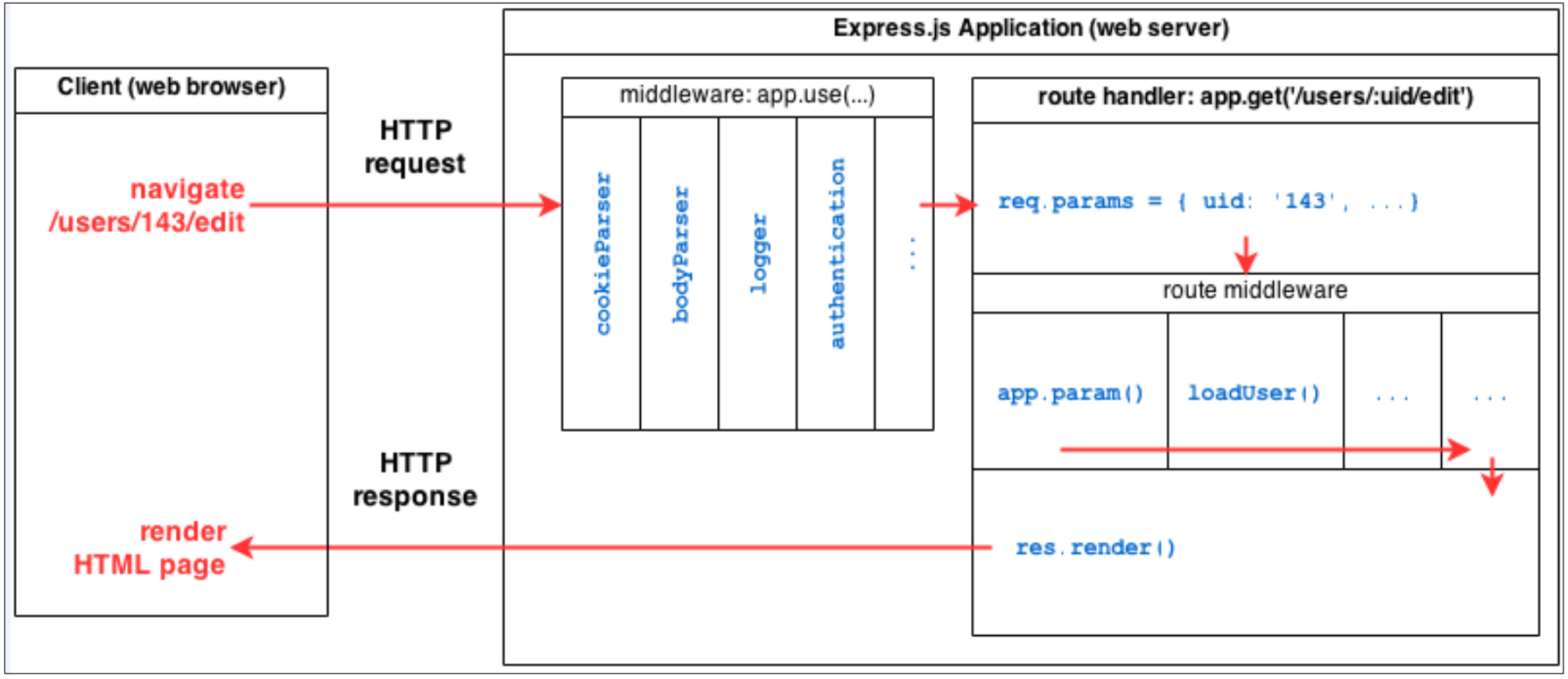
response object properties & methods

Property/Method	Description
`res.jsonp([status|body], [body])`	Sends a JSON object for HTTP response with JSONP support, along with an optional HTTP response code
`res.type(type)`	Sets the media type HTTP response header
`res.format(object)`	Sends a response conditionally, based on the request HTTP Accept header
`res.attachment([filename])`	Sets response HTTP header Content-Disposition to attachment
`res.sendfile(path, [options], [callback])`	Sends a file to the client
`res.download(path, [filename], [callback])`	Prompts the client to download a file
`res.links(links)`	Sets the HTTP Links header
`res.locals`	The object to store variables specific to the view rendering a request
`res.render(view, [locals], callback)`	Renders a view

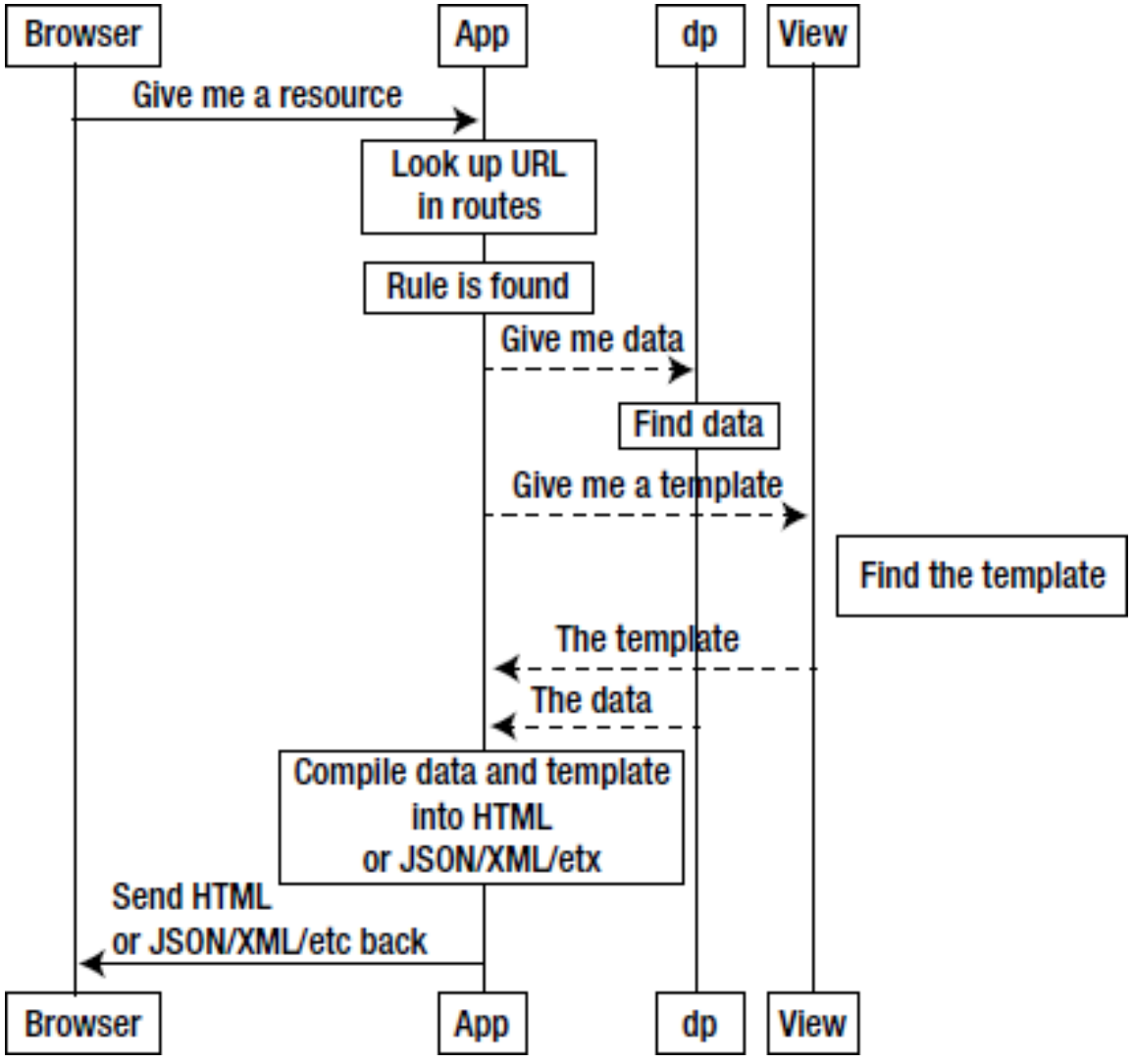
How Express.js works

- **Express.js usually has an entry point, a main file. Most of the time, this is the file that we start with the node command or export as a module. In the main file we do the following**
  - Include third-party dependencies as well as our own modules, such as controllers, utilities, helpers, and models
  - Configure Express.js app settings such as template engine and its file extensions
  - Connect to databases such as MongoDB, Redis, or MySQL (optional)
  - Define middlewares and routes
  - Start the app and Export the app as a module (optional)
- **When the Express.js app is running, it's listens to requests. Each incoming request is processed according to a defined chain of middleware and routes, starting from top to bottom.**

How Express.js works



Request flow in Express



Request flow in Express

- **In Express server the request flow will be :**
  - Route → Route Handler → Template → HTML
- **The route defines the URL schema. It captures the matching request and passed on control to the corresponding route handler**
- **The route handler processes the request and passes the control to a template.**
- **The template constructs the HTML for the response and sends it to the browser.**
- **The route handler need not always pass the control to a template, it can optionally send the response to a request directly.**

Using middleware

- **A middleware is a JavaScript function to handle HTTP requests to an Express app.**
- **It can manipulate the request and the response objects or perform an isolated action, or terminate the request flow by sending a response to the client, or pass on the control to the next middleware.**
- **A middleware can:**
  - Execute any code.
  - Make changes to the request and the response objects.
  - End the request-response cycle.
  - Call the next middleware in the stack.
- **If the current middleware does not end the request-response cycle, it must call next() to pass control to the next middleware, otherwise the request will be left hanging.**

Types of middleware

- **An Express application can use the following kinds of middleware:**
  - Application-level middleware
  - Router-level middleware
  - Built-in middleware
  - Third-party middleware

Express.js Scaffolding

- **To generate a application skeleton for Express.js app, we need to run a terminal command express [options] [dir | appname] the options for which are the following:**
  - -e, --ejs: add EJS engine support (by default, Jade is used)
  - -c <engine>, --css <engine>: add stylesheet <engine> support, such as LESS, Stylus or Compass (by default, plain CSS is used)
  - -f, --force: force app generation on a nonempty directory
  - C:\myapp> express –e -f

Unit Testing, Logging & Debugging

Introduction

- **Unit Testing is nothing but breaking down the logic of application into small chunks or 'units' and verifying that each unit works as we expect.**
- **Unit Testing the code provides the following benefits :**
  - Reduce code-to-bug turn around cycle
  - Isolate code and demonstrate that the pieces function correctly
  - Provides contract that the code needs to satisfy in order to pass
  - Improves interface design
- **Unit testing can be done in 2 ways**
  - Test-Driven Development
  - Test-After Development

Test-Driven Development

- **In Test Driven Development (TDD) automated unit tests are written before the code is actually written. Running these tests give you fast confirmation of whether your code behaves as it should.**
- **TDD can be summarized as a set of the following actions:**
  1. **Writing a test :** In order to write the test, the programmer must fully comprehend the requirements. At first, the test will fail because it is written prior to the feature
  2. **Run all of the tests and make sure that the newest test doesn't pass :** This insures that the test suite is in order and that the new test is not passing by accident, making it irrelevant.
  3. **Write the minimal code that will make the test pass :** The code written at this stage will not be 100% final, it needs to be improved at later stages. No need to write the perfect code at this stage, just write code that will pass the test.

Test-Driven Development

4. **Make sure that all of the previous tests still pass:** If all tests succeeds, developer can be sure that the code meets all of the test specifications and requirements and move on to the next stage.
5. **Refactor code :** In this stage code needs to be cleaned up and improved. By running the test cases again, the programmer can be sure that the refactoring / restructuring has not damaged the code in any way.
6. **Repeat the cycle with a new test :** Now the cycle is repeated with another new test.

- **Using TDD approach, we can catch the bugs in the early stage of development**
- **It works best with reusable code**

Test-After Development

- **In Test-After Development, we need to write application code first and then write unit test which tests the application code.**
- **TAD approach helps us to find existing bugs**
- **It drives the design**
- **It is a part of the coding process**
- **Enables Continual progress and refactoring**
- **It defines how the application is supposed to behave.**
- **Ensures sustainable code**

Behavior-Driven Development

- **Test-Driven Development (TDD) focus on testing where as Behavior-Driven Development (BDD) mainly concentrates on specification.**
- **In BDD we write test specifications, not the tests which we used to do in TDD i.e. no need focus anymore on the structure of the source code, focus lies on the behavior of the source code**
- **In BDD we need to follow just one rule. Test specifications which we write should follow *Given-When-Then* steps**
- **It is very easy to write / read and understand the test specifications, because BDD follows a common vocabulary instead of a test-centered vocabulary (like test suite, test case, test …)**
- **BDD is a specialized version of TDD that specifies what needs to be unit-tested from the perspective of business requirements.**

Behavior-Driven Development

Story: As a registered user, need to login in to the system to access home page. If username or password are invalid, they will stay in login page and the system will shows an error message.

Scenario: Valid Login
Given The user is in login page
And the user enters a valid username
And the user enters a valid password
When the user logs in
Then the user is redirected to Home page

Scenario: Enter an Invalid password
Given The user is in login page
And the user enters a valid username
And the user enters a invalid password
When the user logs in
Then the user is redirected to the Login Page
And the System shows the following message : "Invalid username or password"

Unit Testing in Node

- In Node there are numerous tools / testing frameworks are available to achieve the objective of having well tested quality code.
- Mocha, Chai and SuperTest integrates very well together in a natural fashion
- Mocha is a testing framework which follows Behavior-driven Development (BDD) interface makes our tests readable and make us very clear what is being tested.
- Mocha allows to use any assertion library like (chai, should.js, expect.js)
- Chai is a BDD / TDD assertion library for node and the browser that can be delightfully paired with any testing framework like Mocha.
- SuperTest is a HTTP testing library which has a bunch of convenience methods for doing assertions on headers, statuses and response bodies. If the server is not already listening for connections then it is bound to an ephemeral port, so there is no need to keep track of ports.

Mocha

- **Mocha is a mature and powerful testing framework for Node.js.**
- **It is more robust and widely used. NodeUnit, Jasmine and Vows are the other alternatives.**
- **Mocha supports both BDD interface's like (describe, it, before) and traditional TDD interfaces (suite, test, setup).**
- **We need explicitly tell Mocha to use the TDD interface instead of the default BDD by specifying *mocha -u tdd <testfile>***
  - **describe** *analogous to* **suite**
  - **it** *analogous to* **test**
  - **before** *analogous to* **setup**
  - **after** *analogous to* **teardown**
  - **beforeEach** *analogus to* **suiteSetup**
  - **afterEach** *analogus to* **suiteTeardown**
- **The mocha package can be installed via**
  - *npm install mocha --save-dev*

Steps to debug

- **Install node-inspector as a global module**
  - npm install –g node-inspector
- **There are two steps needed to get you up and debugging**
- **Step – 1: Start the Node Inspector Server**
  - ***node-inspector***
- **Step – 2: Enable debug mode in your Node process**
  - To start Node with a debug flag ***node --debug program.js***
  - To pause script on the first line ***node –debug-brk program.js***