

BUILDING WEB APPS NODE.JS

Build scalable server-side and
networking applications

The Node.js logo is centered in the lower half of the cover. It features the word 'node' in a white, lowercase, sans-serif font, with a green hexagon replacing the letter 'o'. To the right of 'node' is a green hexagon containing the letters 'JS' in white. The background is a dark gray with a complex network of green lines and glowing green hexagons, suggesting a network or server architecture.

node JS

PIYAS DE



Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Building web apps with Node.js

Contents

1	Node.js Installation Tutorial	1
1.1	What is Node.js?	1
1.2	Node Package Manager	1
1.3	Node.js Installation	2
1.3.1	Install Node.js in Ubuntu Linux from Binary	2
1.3.2	Install Node.js in Windows from Binary	3
1.3.3	Build and install Node.js from source code in Ubuntu Linux	11
1.4	Creating a http server in Node.js	12
1.5	Download the Source Code	14
2	Getting Started with Node.js	15
2.1	Introduction	15
2.2	What makes Node any different from the rest?	15
2.2.1	Explanation of source code of a http server creation	16
2.2.2	Explanation of the above node.js code	16
2.3	Event Driven Programming Model	17
2.4	Node.js non blocking I/O	18
2.5	Node.js as a tool	18
2.6	The REPL	18
2.7	Parallel Code Execution	18
2.8	DOM Handling in Node.js	18
2.9	NPM- The Node Package Manager	18
2.10	Understanding Node.js Event Loop	18
2.11	Node.js Event Driven Programming	21
2.12	Asynchronous Programming in Node.js	23
2.12.1	Code snippet for asynchronous programming	23
2.13	Event Emitter in Node.js	23
2.13.1	The .on method	24
2.13.2	The .once method	24
2.14	Creating a Event Emitter	24
2.15	Download the Source Code	25

3	Modules and Buffers	26
3.1	Introduction	26
3.2	Load and Export Modules	26
3.2.1	Load	27
3.2.2	Export	27
3.3	Node.js Buffer Operations	29
3.4	Event Emitter in Node.js	31
3.5	Download the Source Code	31
4	Full application example	32
4.1	Introduction	32
4.2	Application Frontend rendering with EJS	33
4.3	EJS Rendering	34
4.3.1	render (data) method	34
4.4	The Node.js Server	35
4.5	Use of Streaming Functionality	37
4.5.1	The model part - csv handling	39
4.6	Download the Source Code	40
5	Express tutorial	41
5.1	Introduction	41
5.2	Express.js Installation	41
5.3	Express.js Objects	41
5.3.1	The application object	41
5.3.2	The request object	42
5.3.3	The response object	42
5.4	Concepts used in Express	42
5.4.1	Asynchronous JavaScript	42
5.4.2	Middlewares in node.js applications	43
5.5	Definition of Routes	43
5.5.1	How to handle routes in express.js	44
5.5.2	Namespaced Routing	44
5.6	HTTP response in Express	45
5.6.1	Setting the HTTP status code	45
5.6.2	Setting HTTP headers	46
5.6.3	Sending data	46
5.7	Sample web application with NeDB	47
5.7.1	Installation	48
5.7.2	Configuration Code	49
5.7.3	Router Handling Code	49
5.7.4	Angular.js part	50
5.7.5	Angular Template and HTML	52
5.8	Download the Source Code	52

6	Command line programming	53
6.1	Introduction	53
6.2	Utility Programs with Node.js	53
6.3	Command Line and User Interaction	55
6.4	File Handling in Node.js Command Line Program	56
6.5	Publish the Program to NPM	57
6.6	Download the Source Code	58

Copyright (c) Exelixis Media P.C., 2015

All rights reserved. Without limiting the rights under copyright reserved above, no part of this publication may be reproduced, stored or introduced into a retrieval system, or transmitted, in any form or by any means (electronic, mechanical, photocopying, recording or otherwise), without the prior written permission of the copyright owner.

Preface

Node.js is an exciting software platform for building scalable server-side and networking applications. Node.js applications are written in JavaScript, and can be run within the Node.js runtime on Windows, Mac OS X and Linux with no changes.

Node.js applications are designed to maximize throughput and efficiency, using non-blocking I/O and asynchronous events. Node.js applications run single-threaded, although Node.js uses multiple threads for file and network events.

In this book, you will get introduced to Node.js. You will learn how to install, configure and run the server and how to load various modules.

Additionally, you will build a sample application from scratch and also get your hands dirty with Node.js command line programming.

About the Author

Born in Kolkata, India in 1977, Piyas De made a headstrong effort to learn, develop, deliver, teach and share his knowledge on different type of software languages and technologies especially on Java/J2EE and related open source technologies.

Being A Sun Microsystems Certified Enterprise Architect with more than 10 long years of professional IT experience in various areas such as Architecture Definition, Define Enterprise Application, Client-server/ e-business solutions, he possess hands on experience to handle a wide range of database ranging from PostGreSQL, SQL Server7.0/2000, Oracle 8i, 10g to Sybase, MySQL and NoSQL databases like MongoDB.

CMM Level 3 Process orientation proved to be a major turning point for him as Project Manager as it has given him the opportunity to explore various languages or frameworks - to name a few GWT, Struts, Spring, Hibernate, Tiles, Oracle ADF, J2EE (Java), PL/SQL etc.

Some of his career's executed projects are the following:

- subscriptions.abp.in - a media company subscription portal
- healthscribes.com - a doctor's and patient's portal
- Social Media Mashup Project - Revvo (ongoing)
- Health Care Solution for Government Authorites
- NoSQL usage in server creation as per PRODML specification

He learns and writes about different aspects of open source technologies like Angular.js, Node.js, MongoDB, Google DART, Apache Lucene, Text Analysis with GATE and related Big Data technologies in his blog (www.phloxblog.in).

Apart from his professional excellence, he is happily married with Ketaki and has a son named Titas. Also, he is an enthusiast in the field of teaching and a humble book worm who takes immense pleasure reading books not only on technologies but also on humour, suspense, comedy and many more. Impeccable affinity towards knowing the distant corners of technologies became the actual force of penning down fresh technological outlooks.

Chapter 1

Node.js Installation Tutorial

1.1 What is Node.js?

According to nodejs.org site:

“Node.js is a platform built on Chrome’s JavaScript runtime for easily building fast, 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 a platform where we can write server side Javascript and build full-fledged web applications. We can even create web servers based on node.js with a proper security model implemented on top of our application.

Node.js enables programmers to write JavaScript on the server side, which provides access to things like the HTTP stack, TCP, file I/O, and databases.

We will need to use an event-driven, non-blocking I/O model for programming a node.js application as the platform is governed by the above model. Node.js is capable of handling concurrent network connections - so it can be used for data-intensive or real time applications.

Some of the applications that can be built with node.js are:

- Web Applications
- HTTP Proxy based applications
- SMTP Servers used for mail

and other applications which are network intensive.

As we mentioned, all programs built with node.js are actually developed using Javascript. So, in order to understand and work with node.js, we expect that the reader has a basic knowledge of Javascript. All of the applications, which we will develop during this course, will be developed with Javascript.

1.2 Node Package Manager

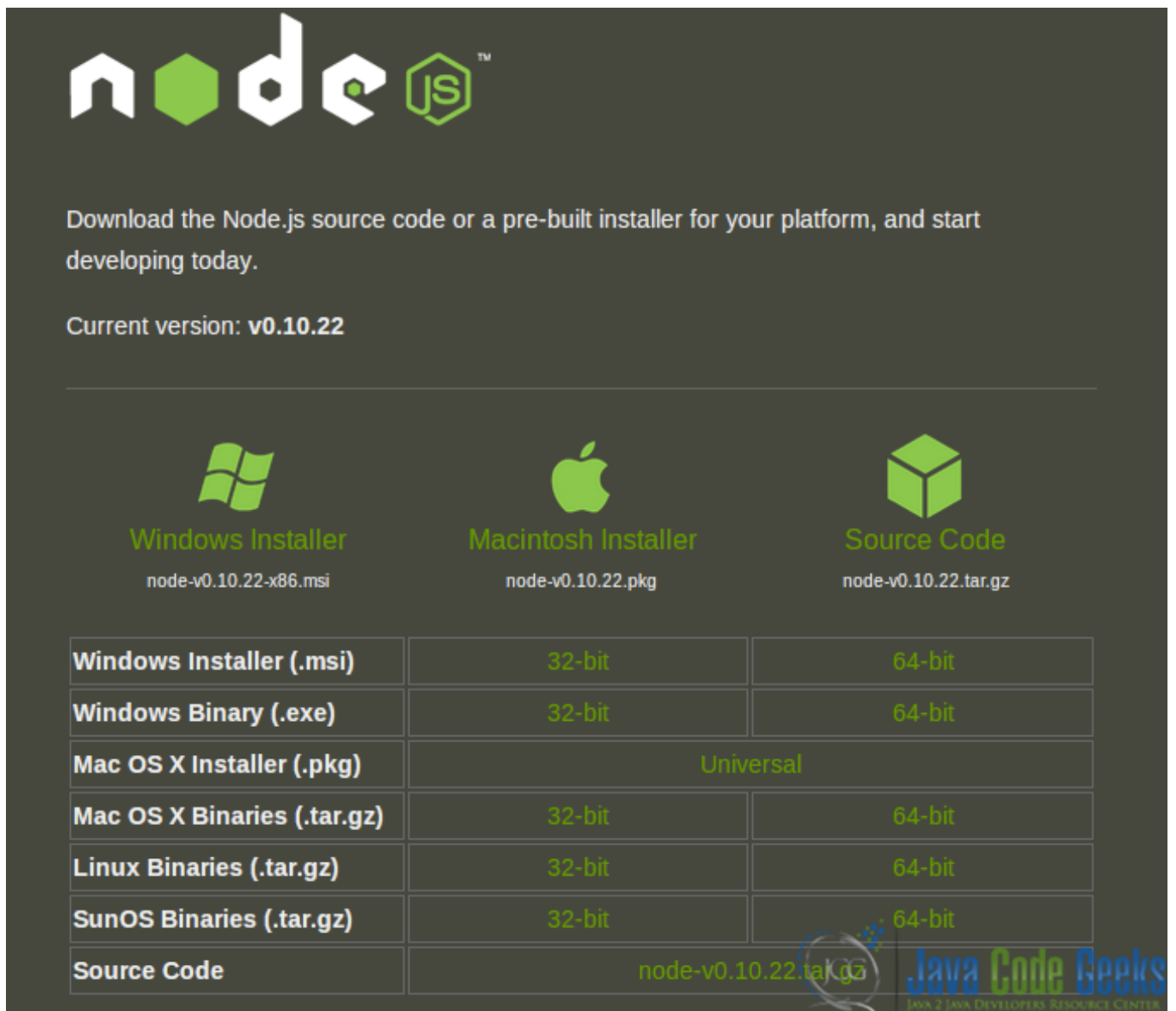
In node.js world, all the libraries for communicating with different applications are maintained as node.js modules. We can install and manage all the modules from the node.js repository using Node Package Manager, or NPM. NPM also allows us to manage modules in a local machine in isolated way, allowing different applications installed in the same machine to depend on different versions of the same module without creating conflicts. Also, NPM allows different versions of the same node.js module to coexist in the same application development environment.

1.3 Node.js Installation

Node.js can be installed on a Windows or Linux Environment.




1.3.1 Install Node.js in Ubuntu Linux from Binary

To install node.js on Ubuntu, we have to go to <http://nodejs.org/download/>.



Download the Node.js source code or a pre-built installer for your platform, and start developing today.

Current version: **v0.10.22**

 **Windows Installer**
node-v0.10.22-x86.msi
  **Macintosh Installer**
node-v0.10.22.pkg
  **Source Code**
node-v0.10.22.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-v0.10.22.tar.gz	

Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Figure 1.1: screenshot

We need to select 32-bit or 64-bit linux binaries as per the requirements of our local machine configuration.

For this example we will download the binary file `node-v0.10.22-linux-x64.tar.gz`.

Now we have to open a terminal and perform the following operations:

First we create a directory named `nodeinstall` in our machine root.

```
sudo mkdir nodeinstall
```

Please note that root privileges are required in order to execute the command.

Now we need to copy the binary using the following command:

```
sudo cp <>/node-v0.10.22-linux-x64.tar.gz /nodeinstall/
```

Go to the folder nodeinstall.

```
cd /nodeinstall/
```

To untar the linux binary, we will use:

```
sudo tar -zxvf node-v0.10.22-linux-x64.tar.gz
```

Now go to /node-v0.10.22-linux-x64/bin folder.

```
cd /node-v0.10.22-linux-x64/bin
```

Now simply run node from command prompt.

```
node
```

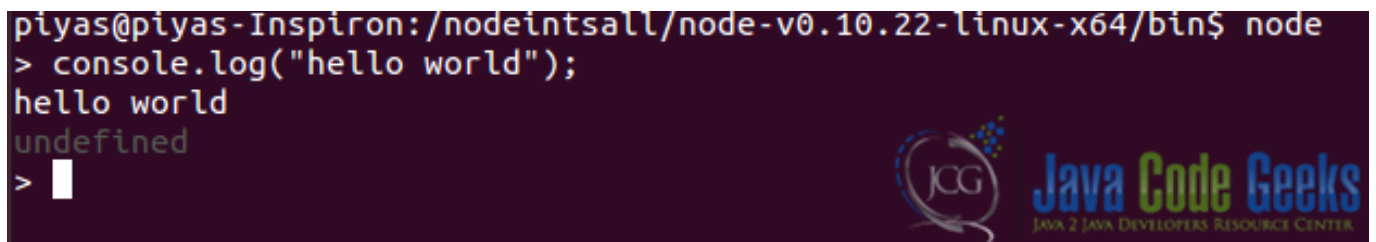
The node.js console will open a node.js interactive console through the Node.js Read-Eval-Print Loop (REPL), to which we can send JavaScript commands.

It will show a prompt with “>”.

Now to quickly check whether everything is fine or not, we will use the following in node.js prompt:

```
console.log("hello world");
```

The Output must match the following screen.



```
piyas@piyas-Inspiron:/nodeinstall/node-v0.10.22-linux-x64/bin$ node
> console.log("hello world");
hello world
undefined
> 
```

Figure 1.2: screenshot

The “hello world” will be prompted in the node.js console. Also we have “undefined” written below the content. Here the REPL will print the value of expression here. Here, since the `console.log` function is returning nothing, the variable value will be printed as “undefined”.

To exit from the node console, we need to press - `ctrl+z` or `ctrl+c` (twice).

1.3.2 Install Node.js in Windows from Binary

Download the Windows installation file from the Node.js site. Double click on the setup file. It will open the installation window.



Figure 1.3: screenshot

Click on Next.

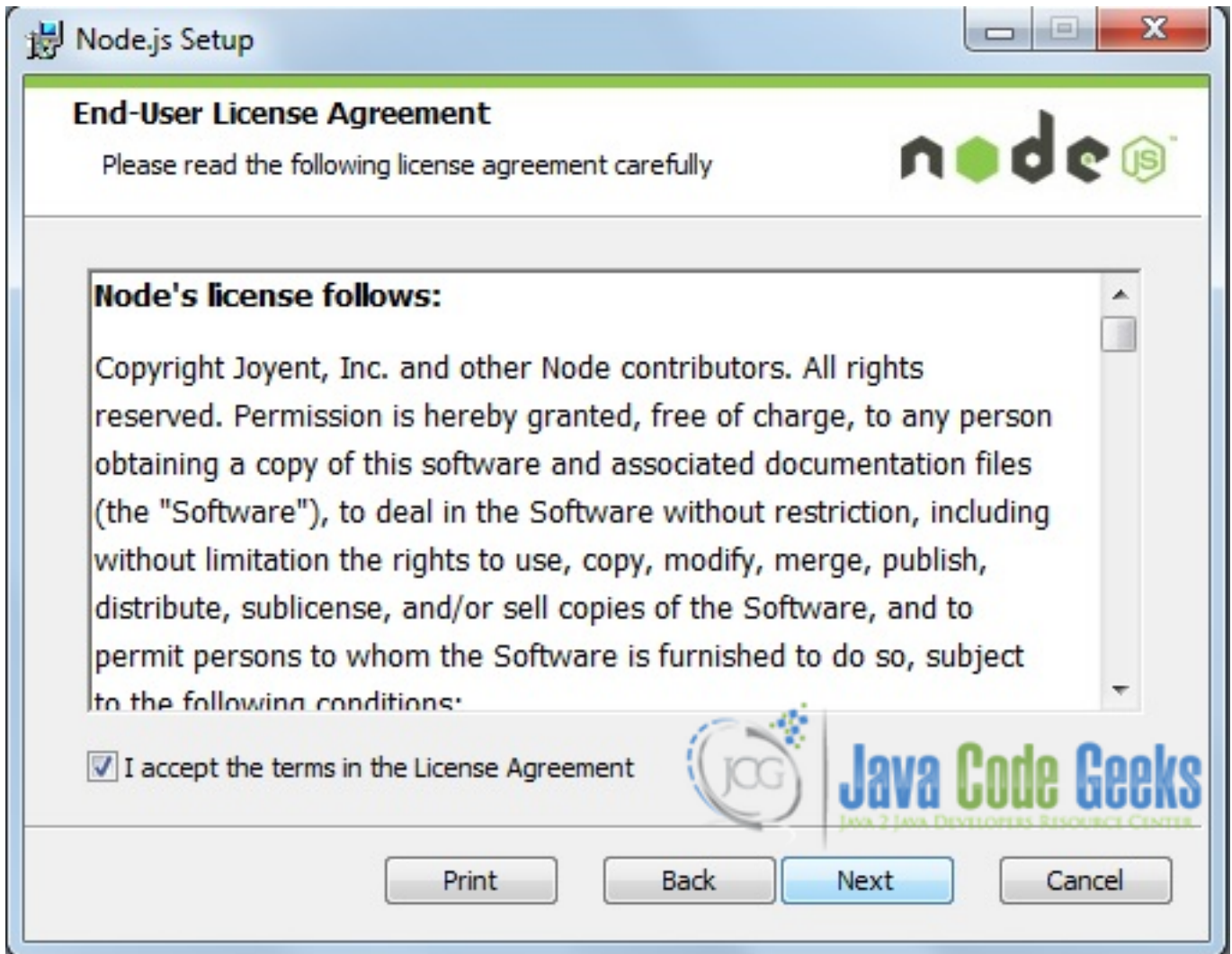


Figure 1.4: screenshot

Click on the check-box of "I accept the terms in the License Agreement", then click Next.



Figure 1.5: screenshot

Provide the installation path where the application will be installed.

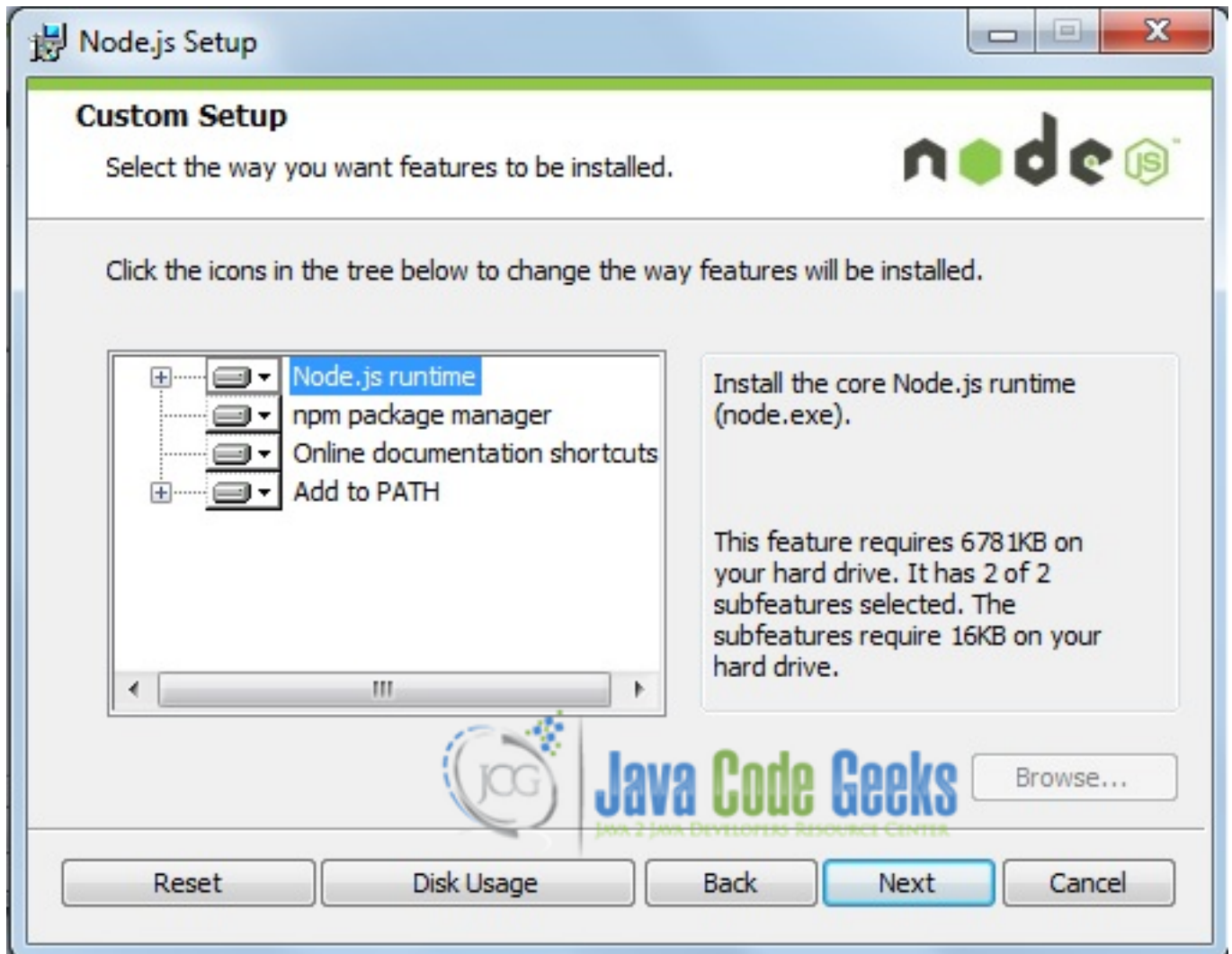


Figure 1.6: screenshot

Click Next.

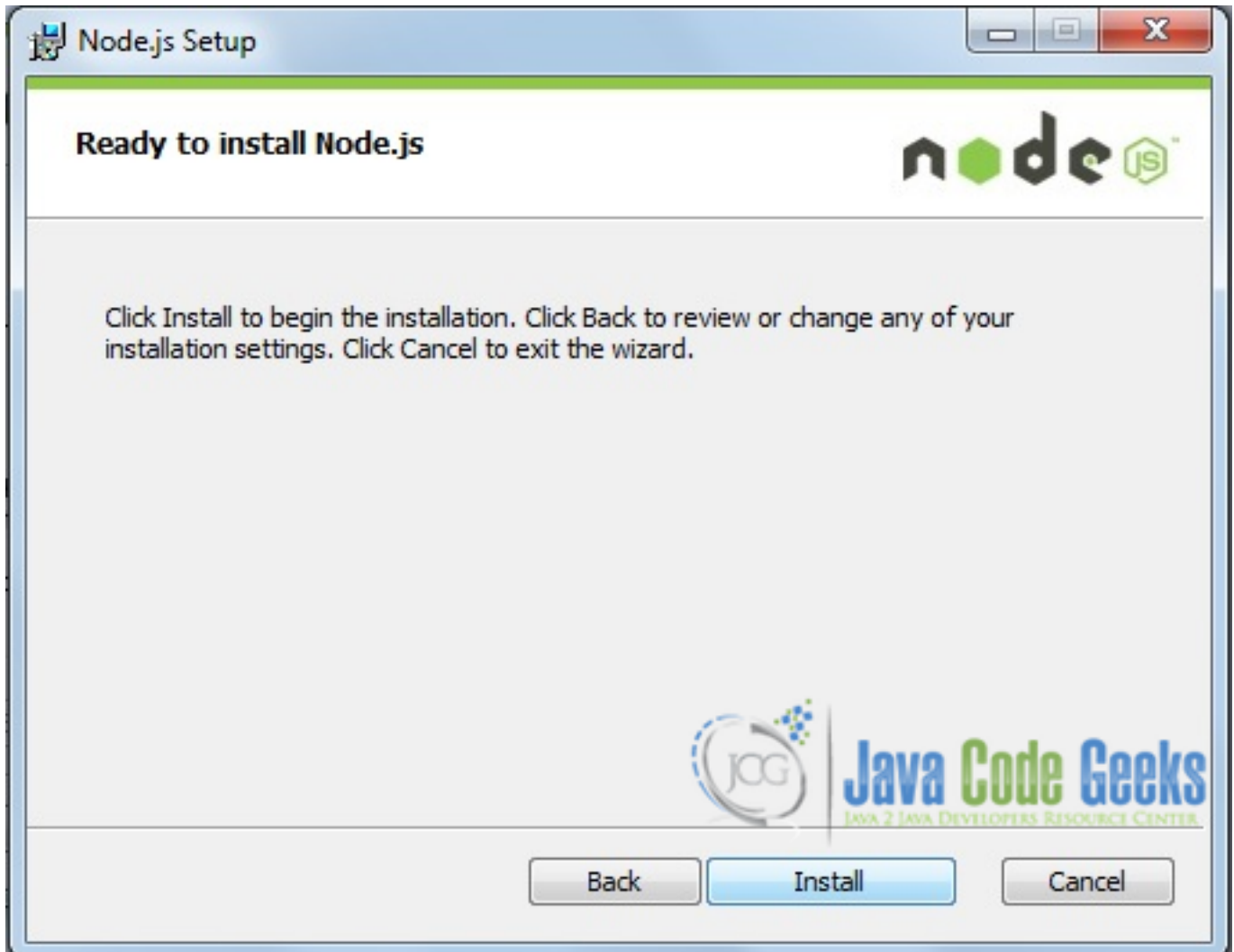


Figure 1.7: screenshot

Click Install.

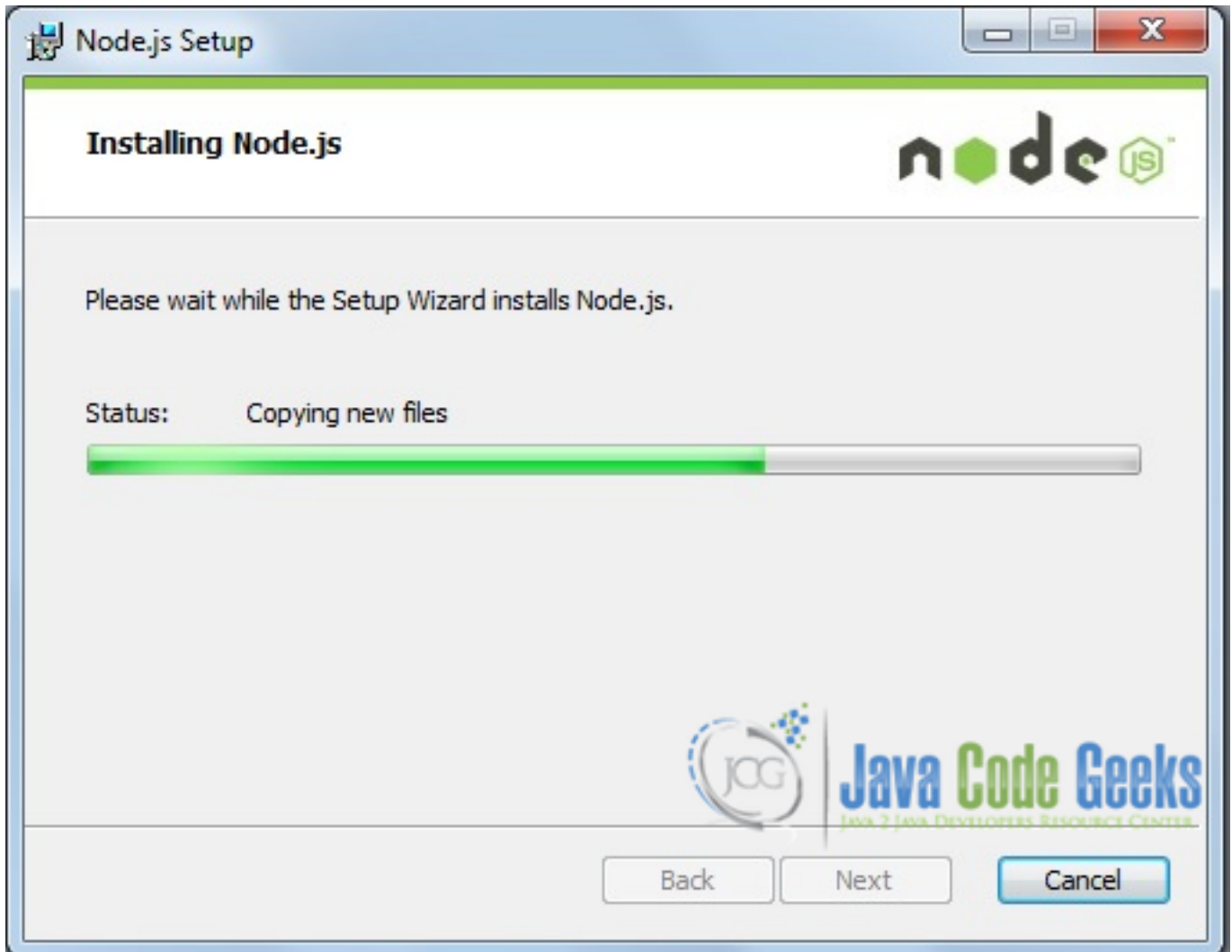


Figure 1.8: screenshot

It will take some time to install. Do not cancel the installation.

After completion we will be provided with the following image:

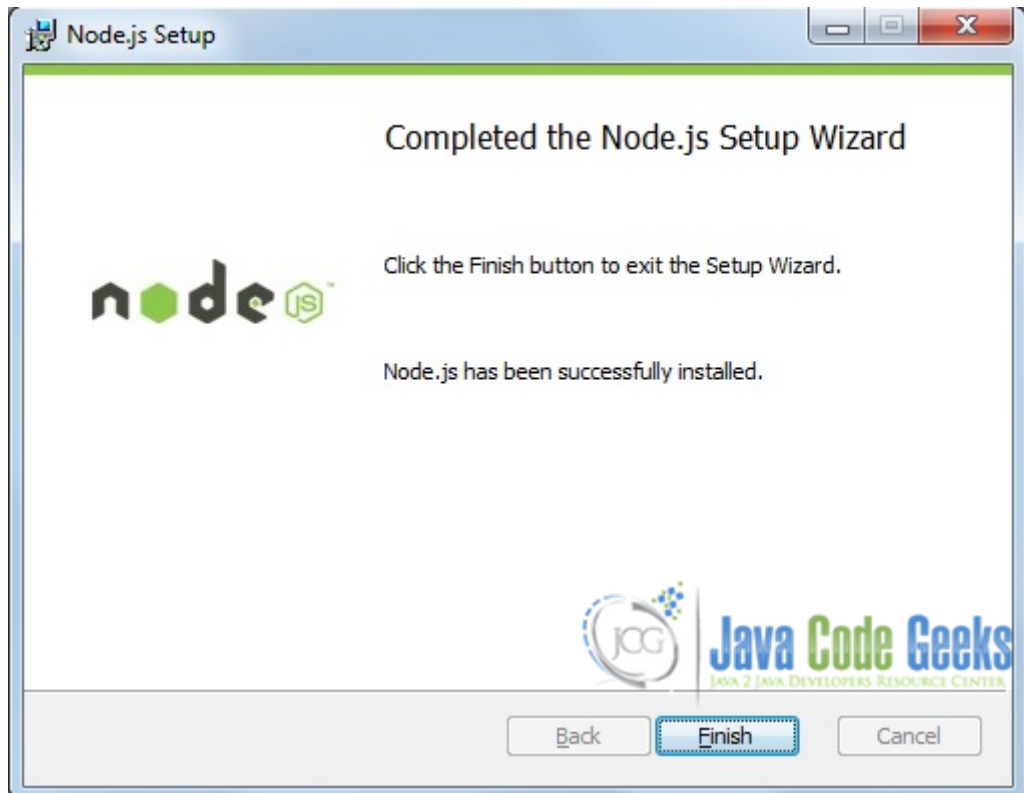


Figure 1.9: screenshot

Click Finish to complete the installation.

Now simply run `node` from command prompt.

```
node
```

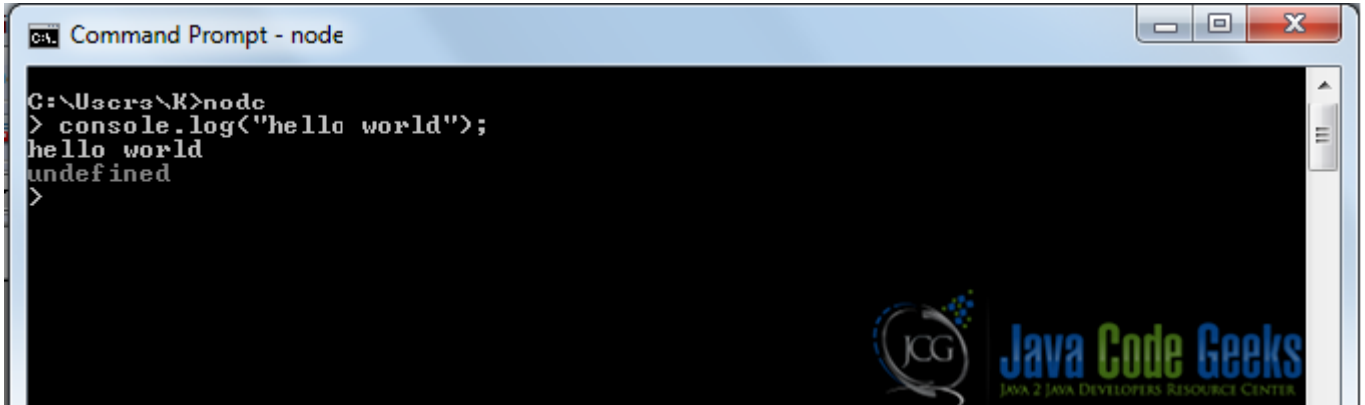
The node.js console will open a node.js interactive console through Node.js Read-Eval-Print Loop (REPL), to which we can send JavaScript commands.

It will show a prompt with ">".

Now to quick check whether everything is fine or not we will use the following in node.js prompt:

```
console.log("hello world");
```

The Output must match the following screen.



```
ca. Command Prompt - node
C:\Users\K>node
> console.log("hello world");
hello world
undefined
>
```

Figure 1.10: screenshot

The “hello world” will be prompted in the node.js console. Also we have “undefined” written below the content. Here the REPL will print the value of expression here. Here as console.log function is returning nothing, the variable value will be printed as “undefined”.

To exit from the node console, we need to press - ctrl+c (twice).

1.3.3 Build and install Node.js from source code in Ubuntu Linux

We need to download the binary file `node-v0.10.22.tar.gz` as per our machine configuration.

Now we have to open a terminal for the following operations:

First we create a directory `nodesourceinstall` in our machine root.

```
sudo mkdir nodesourceinstall
```

Please note that root privileges are required in order to execute the command.

Now we need to copy the binary using the following command:

```
sudo cp <>/node-v0.10.22.tar.gz /nodesourceinstall/
```

Go to the folder `nodesourceinstall`.

```
cd /nodesourceinstall/
```

To untar the linux binary, we will use:

```
sudo tar -zxvf node-v0.10.22.tar.gz
```

Now go to `/node-v0.10.22/` folder.

```
cd /node-v0.10.22/
```

Now the hard work. (Following 3 commands are to be executed one-by-one):

```
sudo ./configure
```

We need to give the root password as needed.

Next is:

```
sudo make
```

The third one is:

```
sudo make install
```

Now we will run node from command prompt:

```
node
```

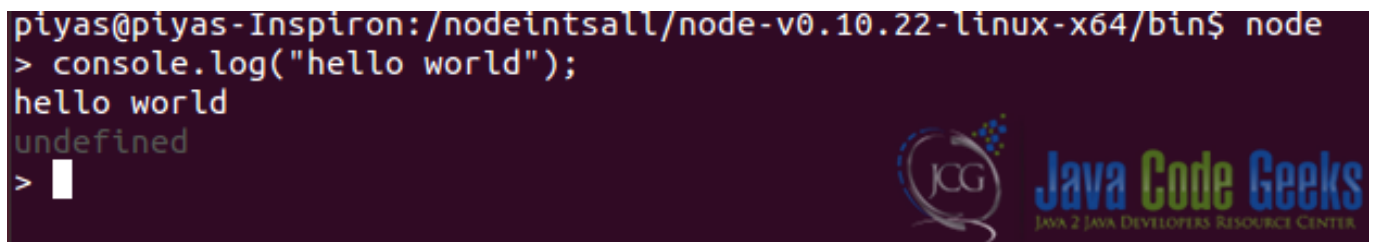
The node.js console will open a node.js interactive console through Node.js Read-Eval-Print Loop (REPL), to which we can send JavaScript commands.

It will show a prompt with ">".

Now to quick check whether everything is fine or not we will use the following in node.js prompt:

```
console.log("hello world");
```

The Output must match the following screen.



```
piyas@piyas-Inspiron:/nodeinstall/node-v0.10.22-linux-x64/bin$ node
> console.log("hello world");
hello world
undefined
> 
```

Figure 1.11: screenshot

The “hello world” will be prompted in the node.js console. Also we have “undefined” written below the content. Here, the REPL will print the value of expression here. Here as console.log function is returning nothing, the variable value will be printed as “undefined”.

To exit from the node console, we need to press - ctrl+z or ctrl+c (twice).

1.4 Creating a http server in Node.js

To get a quick handle on node.js applications, we can write a simple http server in node.js.

Procedures:

- We will create a folder named `nodeapps` in our machine.
- We will create a file named `sampleserver.js` in the folder.

And we will write the following code in the `sampleservers.js` file, as follows:

`sampleserver.js`:

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

function dealWithWebRequest(request, response) {
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello Node.js");
    response.end();
}

var webserver = http.createServer(dealWithWebRequest).listen(8124, "127.0.0.1");
```

```
webserver.once('listening', function() {  
    console.log('Server running at http://127.0.0.1:8124/');  
});
```

Now we will run the following command in the Linux terminal using the node.js executable:

```
node sampleserver.js
```

We will have the following message in the console (Server running at http://127.0.0.1:8124/).

```
piyas@piyas-Inspiron:/nodeapps$ cd nodeapps/  
piyas@piyas-Inspiron:/nodeapps/nodeapps$ node sampleserver.js  
Server running at http://127.0.0.1:8124/  
█
```

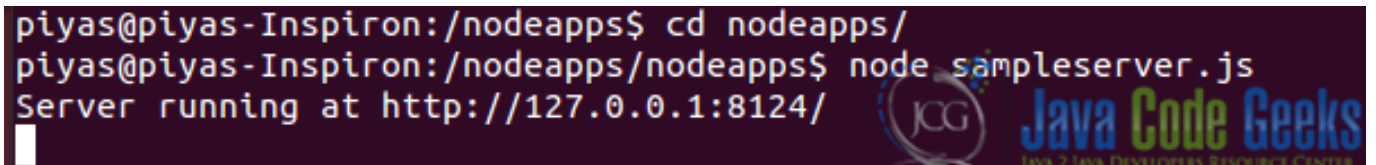


Figure 1.12: screenshot

If we point the web browser to the previous URL, we will have following output:



Figure 1.13: screenshot

Now let's explain the above node.js code.

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

We are using the Node.js http module. We are doing a lookup for the http module in the function call. We are calling the http module and assign the functionality to any variable (here it is "http"). This variable further serves for functional references in next calls.

```
var webserver = http.createServer(dealWithWebRequest).listen(8124, "127.0.0.1");
```

`http.createServer` will create a http server here and will assign it to variable `webserver`. Also, at the end of the closure, the TCP port and server address are given. These define where the server will run. Here, the server address is 127.0.0.1 and the TCP port is 8124.

We're binding the function to the server request event. This event will fire when server receives a HTTP request. This function will be called for every request that this server receives, and we need to pass two objects as arguments: HTTP request and HTTP response. The response object will be responsible for the reply message to the client.

```
function dealWithWebRequest(request, response) {  
    //Some code here  
}
```

This is the callback function, which is passed as argument in `http.createServer(...)` module which is a normal function passing as argument in Javascript.

1. Three inner lines in the function:

```
response.writeHead(200, {'Content-Type': 'text/plain'});  
response.write('Hello Node.js\n');  
response.end();
```

We are writing an HTTP header specifying the content type of the response. So, for this sample server we have set the content type as text/plain in the first line.

Next, we are writing the message with `response.write` and ending the HTTP server response object to render message to the browser. This means the `response.end` will send the message to browser and tell HTTP Protocol to end the response.

```
response.write('Hello Node.js\n');  
response.end();
```

1. After completing the whole work we have the listener setup for the server

```
server.once('listening', function() {  
    console.log('Server running at http://127.0.0.1:8124/');  
});
```

Once the server is listening on that port, the listening event is firing. We have instructed node.js to log a message in that particular listening event.

1.5 Download the Source Code

This was a tutorial of installing Node.js. You may download the source code of this tutorial here: [SampleServer.zip](#)

Chapter 2

Getting Started with Node.js

2.1 Introduction

In recent times, one of the most hyped topics in the web programming world is Node.js. Almost all modern developers have at least heard about this. Some of them have coded and executed their projects using it as well. But still, many of the programming people find it hard to grasp the concepts of node.js programming.

When we introduce Node.js we can simply describe Node as Server-side javascript.

The learnign curve ofor writing Server Side JavaScript applicaiton cab be mitigated of course if we already have experience with client side Javascript. But easiness of development is not the only reason to choose node.js as our server side platform. There are also many others: Node.js is performance oriented and functional code centric.

Also keep in mind that any other full functional server can be build with node.js, like a mail server(SMTP), a server for chat applications and many others.

2.2 What makes Node any different from the rest?

Node.js is a platform built on the V8 javascript engine which is based on an event driven programming model and adopts a non-blocking I/O approach.

V8 is Google's JavaScript implementation that is used in Chrome browser as a runtime. V8 achieves a great speed of executing javascript which is allmost similar to the performance of dynamic languages like Ruby or Python.

Also the closure feature of the javascript language (essentially using callback functions) makes event driven programming even easier to implement.

Node uses a single non-blocking thread with an event loop in order to serve the incoming client requests on the web. Javascript is designed to serve a single threaded event-loop based model in the client side. Additionally, it is used often to serve the dynamic nature of the client side work in different browsers.

See for example the following code:

```
var server = http.createServer(function (request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}).listen(8124, "127.0.0.1");
```

Here, we make a call to the `createServer` method and the anonymous function works as a callback function. The request and response parameters, as well as the `server` variable will be in ready state when the callback function is invoked.

We install the library using the `npm` command like this:

```
npm install <>
```

This will actually download and install the library within the node repository, either local or global. Then, we can use the methods of the library as per our requirement in the node.js application development. We generally need to add the library using the require function. This function returns the object of the particular library (in node convention this is called module) as a variable which we declare in our application source code file (in node convention this is called namespace).

See how this is achieved in the code below:

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

var server = http.createServer(function (request, response) {
    // Some code here
}).listen(8124, "127.0.0.1");
```

The variable `http` will hold the full functionality of the node.js http server, which calls the `createServer` method and returns the object in the `server` variable.

2.2.1 Explanation of source code of a http server creation

We create a file with name `sampleserver.js` as follows:

`sampleserver.js`:

```
// Creation and running of the Server
var http = require('http');

function dealWithWebRequest(request, response) {
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello Node.js");
    response.end();
}

var webserver = http.createServer(dealWithWebRequest).listen(8124, "127.0.0.1");

webserver.once('listening', function() {
    console.log('Server running at http://127.0.0.1:8124/');
});
```

Now we will run the following command (in a Linux terminal) using the node.js executable.

```
node sampleserver.js
```

We will have the following message in the console:

```
Server running at http://127.0.0.1:8124/
```

2.2.2 Explanation of the above node.js code

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

We are using the Node.js http module. We are doing a lookup for the HTTP module in the function call. We are actually calling the http module and assign its functionality to any variable (here it is `http`). This variable further serves for functional references in any subsequent calls.

```
var webserver = http.createServer(dealWithWebRequest).listen(8124, "127.0.0.1");
```


`http.createServer` will create a http server and assign it to the variable `webserver`. Also note that at the end of the closure, the TCP port and server address are provided. These parameters define where the server will execute. Here the server address is 127.0.0.1 and the TCP port is 8124.

With the above lines, we're binding the function to the server request event. This event will fire when the server will receive an HTTP request. This function will be called for every request that this server receives, and we need to pass two objects as arguments:

- HTTP request
- HTTP response.

The response object will be responsible for the reply message to the client.

```
function dealWithWebRequest(request, response) {  
  
    //Some code here  
  
}
```

This is the callback function, which is passed as argument in `http.createServer(...)` module. We use a normal function passing as an argument in javascript.

1. Let's now see the following three inner lines in the function:

```
response.writeHead(200, {'Content-Type': 'text/plain'});  
response.write('Hello Node.js\n');  
response.end();
```

We are writing the HTTP header specifying the content type of the response. For this sample server we have set the content type as `text/plain` (see first line).

Next, we are writing the message with `response.write` and "ending" the HTTP server response object to render the message to the browser. This means the `response.end` will send the message to browser and instruct the HTTP Protocol to end the response:

```
response.write('Hello Node.js\n');  
response.end();
```

1. After completing the whole work we have the listener setup for the server:

```
server.once('listening', function() {  
    console.log('Server running at http://127.0.0.1:8124/');  
});
```

Once the server is listening on that port, it will listen for event that are firing. We have defined a log message for that particular listening event. The message will fire only the first time, when the server will emit the message and so the `server.once` function will have been called.

In Node.js all files have their own module level scope to which all global declarations belong. Note that in Node.js, it is not possible to call a function that blocks for any reason. At least most of the functions residing in the various modules are designed based on this principle.

2.3 Event Driven Programming Model

As we have mentioned, Node.js programming is an event driven programming model. Here, the execution of a function or a set of code is driven by events.

Here are some events that are of interest:

- when someone clicks on the mouse to press one button, some functionality is triggered in the application or
- when the remote control button is pressed, the TV Channel changes

These are example of real world callbacks in the case of some event execution.

2.4 Node.js non blocking I/O

A web request comes in the node.js web server and the web server accepts this request and routes it to a listener object to process the response. Also this web server will remain ready to accept any new web requests. The previous response handling will remain in a queue in order to perform the rest of operations. The node.js environment takes care of preparing the web response which again can be a database call from the actual application.

2.5 Node.js as a tool

Node.js can also be used as a command line tool. We can get the Windows Distribution (MSI/EXE) or the Linux/ Mac Binary to [install node.js](#). Also we can get the node.js source code to build the binaries from it.

After installing Node.js, when we write `node` in a command prompt, we will be presented with an interactive javascript console for it, where we can execute various node.js commands. We can run node.js based programs with `node ourprogram.js` as a usual convention.

2.6 The REPL

In node.js the REPL represents for Read-Eval-Print-Loop which is a simple program that accepts tasks, evaluates the input, and prints their results back. In the node.js interactive console, we can execute most of the commands without creating separate files and it prints the results as we expect.

2.7 Parallel Code Execution

In node.js every workable unit will run in parallel, except the code that runs in only one process. The main process gets the user request and routes it to various workable modules in order to be executed and to be acknowledged by those corresponding modules. The main process in node.js constantly runs a loop after delegating the work to different callback handlers for various modules as necessary. The main process always checks for any new request to be served. Also when the completion acknowledgement comes from different sub-modules, it shows the outcome (in web convention we can call it response or deferred response).

2.8 DOM Handling in Node.js

Nowadays, there are helper libraries available to manipulate DOM from node.js modules. Here is an useful [link](#). But as we understand, the DOM Manipulation is still a client side concept which is not the main focus of the node.js platform.

2.9 NPM- The Node Package Manager

Node.js is having an ever growing repository of different utilities and server side tasks. These modules can be installed through the Node Package Manager (NPM) in our local development environmen. The general syntax for installing a new module is:

```
npm install modulename.
```

2.10 Understanding Node.js Event Loop

In a traditional web programming model, such as in a JEE Environment, each web request is served by one thread. This thread usually has less CPU overhead than a standalone process, but all operations in that thread are synchronous. Usual, the thread

functionality is not implemented with asynchronous programming in mind. As those models are synchronous, the application behaves like a blocking operation to the end user and can also be memory-expensive.

Within a web request-response cycle, most of the the time is spent in order to complete the I/O Operations. So these I/O operations can have a significant performance impact on the application, especially if the application is processing thousand of hits per minute or more.

And here is how the node.js non-blocking programming models works efficiently. In contrast to the traditional web programming model, node.js platform works in one single process which handles the web requests.

But all the IO operations and the other function calls are asynchronous in nature. They are "waked" by a callback listener from the node.js main process and they run each of the calls in different threads (Child Processes).

In the main process, the node.js server call is never blocked as this process only delegates the IO tasks to the sub processes (worker threads) and the response to the client whenever the sub processes are completed. Additionally, the main process continuously loops over to handle new web requests and to respond to the client to handle over the results as found from the relatively long running sub-processes.

So, all the works in the main process in node.js are notification/event driven and continuous in nature. That is the reason why the node.js main process is known as the node.js event loop.

All the child threads are handled by the node.js platform itself and the application programmer does not need to know about the inner details (though conceptually knowing the architecture of the platform will definitely help the programmer to code more efficiently).

We can derive a flow as follows:

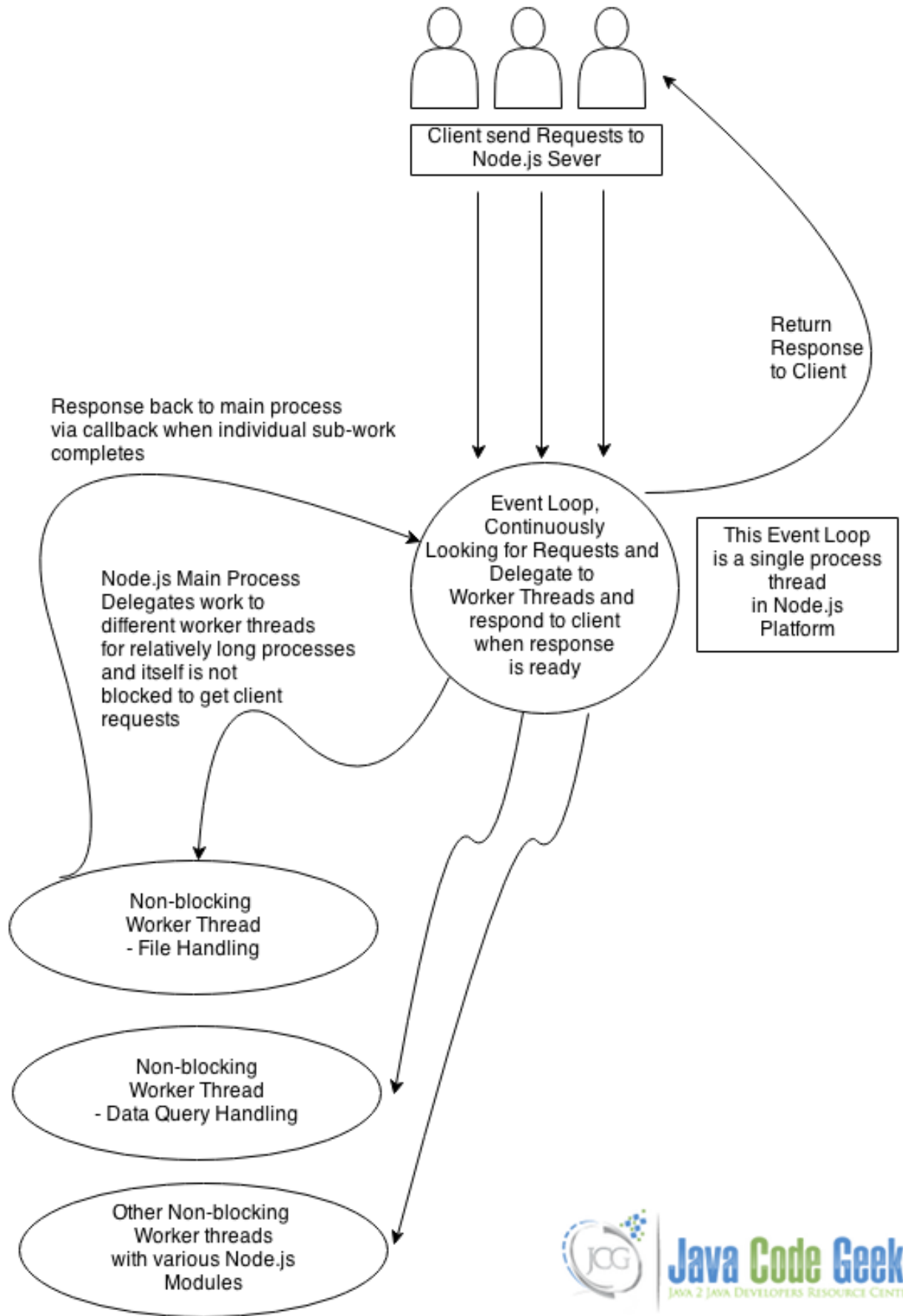


Figure 2.1: screenshot

2.11 Node.js Event Driven Programming

According to wikipedia:

"In computer programming, event-driven programming (EDP) or event-based programming is a programming paradigm in which the flow of the program is determined by events, e.g. sensor outputs or user actions (mouse clicks, key presses) or messages from other programs or threads."

Event driven programs are driven by two factors - one is the Event Occurance or Event Invocation/Generation and another is some unit of work executed upon listening to the Event.

Now, from the node.js perspective, we need to understand how the platform approaches the event driven programming model. The underlying architecture of the node.js platform is different from the traditional runtimes for java, ruby, php etc.

Let see the following code:

```
var file = fileSystem.read("big.txt");
console.log(file.toString());

var takeAnotherFile = fileSystem.read("anotherbigfile.txt");
console.log(takeAnotherFile.toString());
```

Here the file can be very big in size and it may take some time to read in memory. After the reading of the content in memory is completed, the application will proceed to the next execution line where the file contents will be shown. Please note, this samples does not actually belong to some language, but is used for illustration purposes only.

After the above operation, the program will continue to the next file, also reading it and so on. It is thus clear that the next lines of code will "wait", until the execution of the previous line is complete.

In node.js, there is completely different approach to solve the wait-until-previous-work-complete scenario. Here is an introduction to the "asynchronous callbacks per event" approach, used like in the following code:

```
fileSystem.read("big.txt", function(data) {
    console.log(data.toString());
});

fileSystem.read("anotherbigfile.txt", function(anotherdata) {
    console.log(another.toString());
});
```

So, by providing the `fileSystem.read` function and making it asynchronous in nature, the application will show the content of the file when reading is complete and it will not wait until the content is completely ready. Instead, it will go proceed to the next execution, i.e. another file read.

Here, the sub-processes that are delegated from the main process will work in self-contained threads. At the same time, the main process will wait for new events, such as requests or notifications from the sub processes about work being completed (this is the famous Event Loop of the node.js platform).

In general, the architecture for Node.js is as follows:

- Receive all of the event generation notification with attached Data.
- There should be a central unit which decides the delegation sequence for the various sub-processes/handlers.
- Handlers which run in self-contained threads, which execute the delegated work unit separately from the main process and notify it about the work completion.

Following is a flow to describe the above:

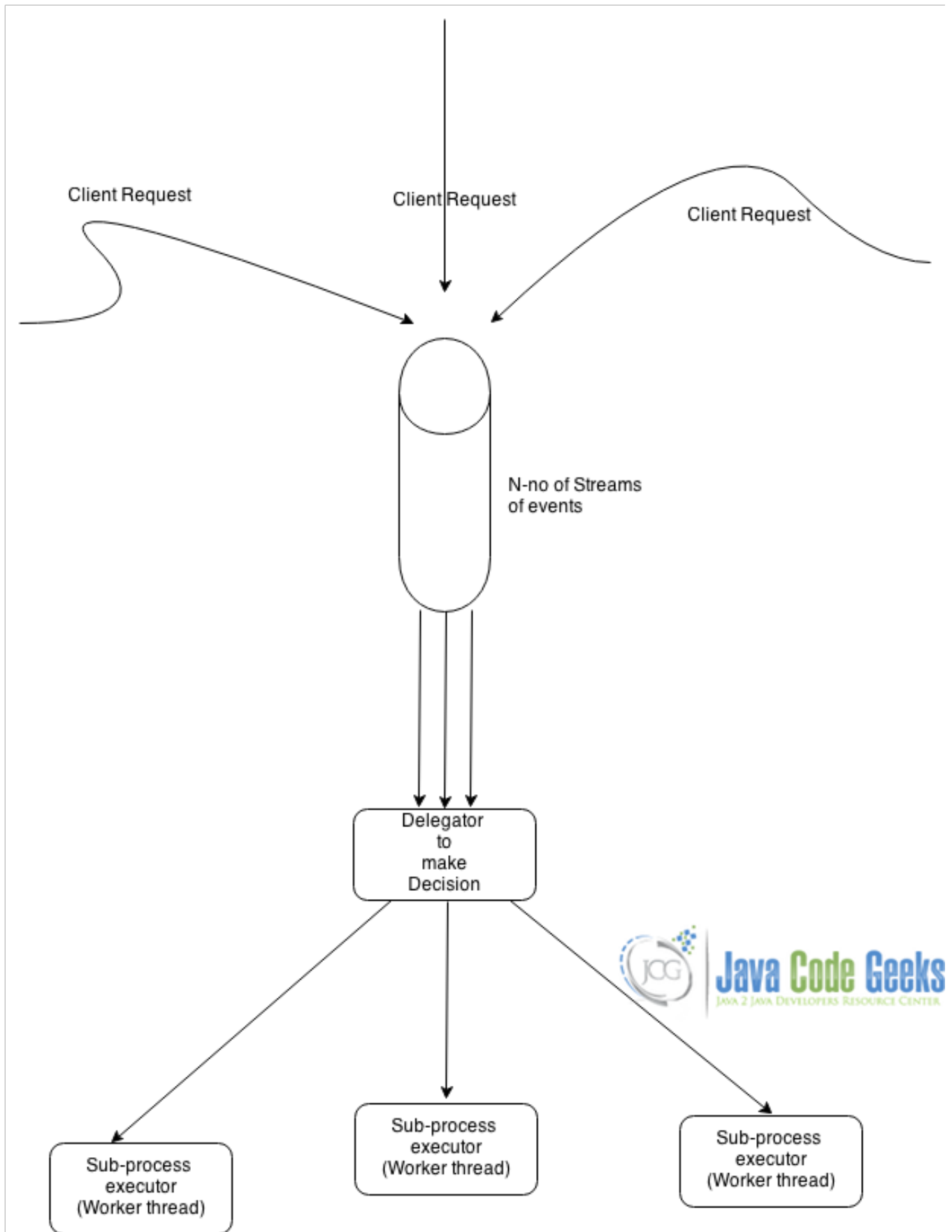


Figure 2.2: screenshot

2.12 Asynchronous Programming in Node.js

In the general context of programming, programmers generally think of code which will execute serially. But this approach is not a good solution for an application where users are interacting with it in real time.

IN sequential execution of code, any I/O operations (usually calls to the various operating system specific functions or network specific functions) will cause the code block to wait. The particular operation will have to be completed in order to proceed. The code will block and eventually the application will seem as a halted process to the end user, preventing him/her to operate further.

The Node.js main process, which continuously listens for server requests, runs as a single process and delegates the sub-operations to the appropriate data handlers for execution and then "listens" for any notification from the sub-process regarding the completion of the assigned work. The main process will never be blocked for any sub-operations from the handlers. So this approach is truly an asynchronous operation.

But please keep in mind that not all functions in node.js are asynchronous. For example, some of the core functions related to file handling are utilizing functions which are synchronous in nature. (Example: `readFileSync` function in `FileSystem` module).

2.12.1 Code snippet for asynchronous programming

Here the single threaded node.js will behave as multi-threaded assigning tasks to the corresponding worker threads. The pattern here is to execute the work from a callback function. The general format for the operation will be like this:

(The following code will execute without waiting for the the callback function response)

```
var callback = function() {
    // return to the response
};
thisIsAsynchronousCall(callback);
```

With this approach, the caller function will need to pass the code to the callee function.

But there is also another promising approach. Here the callee function will not need to know about the callback. And it will remain as a separate function with self-contained code. So, in this method we will have code like this:

```
var callback = function() {
    // return to the response
};

Promise.when(thisIsAsynchronousCall())
    .then(callback);
```

Here the callback function will never be passed to the asynchronous `thisIsAsynchronousCall()` function. The functions whether synchronous or asynchronous, will be handled by the "Promise" concept.

A popular implementation of the promise concept in node.js is Q.

2.13 Event Emitter in Node.js

On Node many objects can emit events. To have access of the events module, we need to install it accordingly. The installation command will be:

```
npm install events
```

2.13.1 The .on method

We can listen for these events by calling one of these objects' `on` method, passing in a callback function. For example, a file `ReadStream` can emit a `data` event every time there is some data available to read.

Here is the code example:

```
var fs = require('fs'); // get the fs module
var readStream = fs.createReadStream('/etc/passwd');

readStream.on('data', function(data) {
    console.log(data);
});

readStream.on('end', function() {
    console.log('file ended');
});
```

Here we are binding to the `readStream`'s `data` and `end` events, passing in callback functions to handle each of these cases. When one of these events happens, the `readStream` will call the callback function.

There are 2 ways of doing the above operation:

- Pass in an anonymous function.
- Pass a function name for a function available on the current scope, or a variable containing the function.

2.13.2 The .once method

When we want the callback method to be called only once, we use this approach in our work:

```
server.once('connection', function (stream) {
    console.log('We have our first call!');
});
```

This will have the same effect as calling the function and then removing the relevant listener:

```
function connectAndNotify(stream) {
    console.log('We have our first call!');
    server.removeListener('connection', connectAndNotify);
}
server.on('connection', connectAndNotify);
```

The `removeListener` function takes the event name and then removes it from the current context of code execution. To remove all Listeners from the current context we should use `removeAllListeners` function.

2.14 Creating a Event Emitter

We have to use the `events` package in `node.js`.

```
var EventEmitter = require('events').EventEmitter,
    util = require('util');
```

Now the Costructor:

```
var eventEmitterClass = function() {
    console.log("The Class Constructor Example");
}

util.inherits(eventEmitterClass, EventEmitter);
```


`util.inherits` will set up the prototype chain so that the EventEmitter prototype methods will be available in `eventEmitterClass` instances.

With this way, instances of `eventEmitterClass` can emit events:

```
eventEmitterClass.prototype.emitMethod = function() {
  console.log('before the emitevent');
  this.emit('emitevent');
  console.log('after the emitevent');
}
```

Here we are emitting an event named `emitevent`.

Now clients of `eventEmitterClass` instances can listen to `emitevent` events like this:

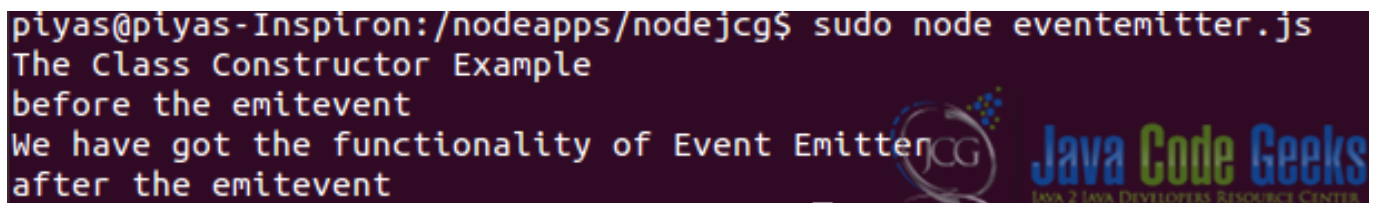
```
var evtEmitInstance = new eventEmitterClass();

evtEmitInstance.on('emitevent', function() {
  console.log('We have got the functionality of Event Emitter');
});
```

Now the calling part from the instance of the class:

```
evtEmitInstance.emitMethod();
```

Here we have got the custom event from the class and work within the event.



```
piyas@piyas-Inspiron:/nodeapps/nodejcg$ sudo node eventemitter.js
The Class Constructor Example
before the emitevent
We have got the functionality of Event Emitter
after the emitevent
```

Figure 2.3: screenshot

2.15 Download the Source Code

The source code for the file is attached here: [Node_Introduction.zip](#)

So, in this article we have covered basic node.js concepts and some of the advanced topics. We will discuss again about those in our next lessons.

Chapter 3

Modules and Buffers

3.1 Introduction

To build structural applications adopting different patterns in different files, node.js provides us with the option for a module based system.

In node.js, we add variables or functions to the object *module.exports*. By invoking the *require* function in a script which uses the module, will return the corresponding object (variables, functions etc ...)

- For a module: *module.exports* points to an object.
- For a script: *require("module-filename")* returns that object.

There are two types of node.js for the module interaction:

1. When node.js does not have a relative hint about the file location.

Example:

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

Here node.js looks for the core module named `http` and returns that module. This module should have been stored in the global repository space of our node.js application.

Now, for the non-core modules, node.js will look into the folder structure and try to find out a folder name called `node_modules` and then find the module reference from that module. In other words, it will look for a required file named as per the required file reference with a javascript extension (`.js`).

1. Now when node.js works with a single file in some folder.

Example:

```
var http = require('./filename');
```

Here node.js has the option to search for both `filename.js` and `index.js` - where the actual javascript file is referenced.

3.2 Load and Export Modules

In the node.js platform, there are requirements to have utility functions and common functions. But by default, all variables, functions, classes and member function of classes are accessible within a particular file only.

Thus, some function declared within one file can not be accessed from another js file. As per definition, related node.js functionalities are described as modules. By default, when we write javascript functions within modules, they are accessed within that class and they are private to that module only, i.e. no other module or file can access these elements (variables, functions etc.). However, there are ways to expose those functions, variables or classes as needed.

3.2.1 Load

In node.js, different external resources can be loaded in another module with the following syntax:

```
var checkData = require('./validatorutil');
```

Here, the *require* keyword is used to load an external module in the working module and the return type is assigned to a variable.

3.2.2 Export

To expose functions and classes, we use `module.exports` and export those as per our requirements.

In the function below, we have used the validator library of node.js.

Check valid Email Address:

```
var checkEmail = function(value) {
  try {
    check(value).isEmail();
  } catch (e) {
    return e.message; //Invalid integer
  }
  return value;
};

module.exports.checkEmail = checkEmail;
```

To use this in our required file we have written them as:

```
var checkData = require('./validation');

console.log("In testvalidation -->" + checkData.checkEmail("piyas.de@gmail.com"));

console.log("In testvalidation -->" + checkData.checkEmail("piyas.de@gmailcom")); // Return ←
error
```

Some other useful functions that we can use for our daily tasks are:

```
// Chcek minimum length

var checkMinLength = function(value, len) {

  try {
    check(value, 'Please specify a minimum length of %1').len(len);
  } catch (e) {
    return e.message;
  }
  return value;
};

//Check Maximum length

var checkMaxLength = function(value, lenmax) {
  try {
    check(value, 'Please specify a maximum length of %2').len(0, lenmax);
  } catch (e) {
    return e.message;
  }
  return value;
};
```

```
//Check boundary length

var checkBoundaryLength = function(value, lenmin, lenmax) {
  try {
    check(value, 'The message needs to be between %1 and %2 characters long (you passed ←
      "%0"') .len(lenmin, lenmax);

  } catch (e) {
    return e.message;
  }
  return value;
};

//Check Numeric

var checkNumeric = function(value) {
  try {
    check(value).isNumeric();
  } catch (e) {
    return e.message;
  }
  return value;
};

//Check AlphaNumeric

var checkAlphaNumeric = function(value) {
  try {
    check(value).isAlphanumeric();
  } catch (e) {
    return e.message;
  }
  return value;
};

//Check LowerCase

var checkLowerCase = function(value) {
  try {
    check(value).isLowercase();
  } catch (e) {
    return e.message;
  }
  return value;
};

//Check UpperCase

var checkUpperCase = function(value) {
  try {
    check(value).isUppercase();
  } catch (e) {
    return e.message;
  }
  return value;
};

module.exports.checkMinLength = checkMinLength;
module.exports.checkMaxLength = checkMaxLength;
module.exports.checkBoundaryLength = checkBoundaryLength;
module.exports.checkNumeric = checkNumeric;
```

```
module.exports.checkAlphaNumeric = checkAlphaNumeric;
module.exports.checkLowerCase = checkLowerCase;
module.exports.checkUpperCase = checkUpperCase;
```

And their use in the working file, respectively:

```
console.log("In testvalidation -->" + checkData.checkMinLength("abc", 2));
console.log("In testvalidation -->" + checkData.checkMinLength("abc", 4)); // Return error
console.log("In testvalidation -->" + checkData.checkMaxLength("abc", 2)); // Return error
console.log("In testvalidation -->" + checkData.checkMaxLength("abc", 4));
console.log("In testvalidation -->" + checkData.checkBoundaryLength("abc", 2, 4));
console.log("In testvalidation -->" + checkData.checkBoundaryLength("abc", 4, 6)); // Return ↔
error
console.log("In testvalidation -->" + checkData.checkNumeric("12"));
console.log("In testvalidation -->" + checkData.checkNumeric("ABC")); // Return error
console.log("In testvalidation -->" + checkData.checkAlphaNumeric("_!")); // Return error
console.log("In testvalidation -->" + checkData.checkAlphaNumeric("A23"));
console.log("In testvalidation -->" + checkData.checkLowerCase("lower"));
console.log("In testvalidation -->" + checkData.checkLowerCase("Lower")); // Return error
console.log("In testvalidation -->" + checkData.checkUpperCase("UPPER"));
console.log("In testvalidation -->" + checkData.checkUpperCase("upPeR")); // Return error
```

This is just the starting point to understand the load and export functionality in node.js.

You may [download](#) the associated Source code to mess around with.

3.3 Node.js Buffer Operations

Pure javascript is not efficient in handling binary data. For this reason, Node.js has a native layer implementation for handling binary data, which is a buffer implementation with the syntax of javascript. In node.js, we utilize buffers in all operations that read and write data. Thus, Node actually provides an interface for handling binary data in a quite efficient way.

Some points to be noted for the node.js buffer implementation:

- A Buffer can not be resized.
- Raw data from the transport layer are stored in buffer instances.
- A Buffer corresponds to raw memory allocation outside the V8 javascript engine.

Now let's examine the syntax:

- To create a buffer for utf-8 encoded string by default:

```
var buf = new Buffer('Hello Node.js...');
```

- To print the buffer, we can write:

```
console.log(buf);
```

- To print the text, as we have entered, we have to write:

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

- To create a buffer of pre-allocated size, we write:

```
var buf = new Buffer(256);
```

- Also we can store a value in each of the pre-allocated arrays with the following syntax:

```
buf[10] = 108;
```

- We can assign encoding while creating the buffer or while printing it by using:

```
var buf = new Buffer('Hello Node.js...', 'base64');
```

or

```
console.log(buf.toString('base64'));
```

- A buffer can be sliced to small buffers with the following syntax:

```
var buffer = new Buffer('this is a good place to start');
```

```
var slice = buffer.slice(10, 20);
```

Here the new buffer variable `slice` will be populated with "good place" which starts from 10th byte and ends with 20th byte of the old buffer.

- Also to copy from a buffer to another buffer variable we write:

```
var buffer = new Buffer('this is a good place to start');
```

```
var slice = new Buffer(10);
```

```
var startTarget = 0, start = 10, end = 20;
```

```
buffer.copy(slice, startTarget, start, end);
```

Here the values will be copied from old buffer to new buffer.

- To copy the whole buffer with any value, we can use the *fill* method:

```
var buffer = new Buffer(50);
```

```
buffer.fill("n");
```

- `buf.toJSON()` returns a json representation of the Buffer object, identical to a json array.

```
var buf = new Buffer('test');
```

```
var json = JSON.stringify(buf);
```

```
console.log(json);
```

Here `json.stringify` is called implicitly to maintain the json representation.

There is in-depth documentation for the Buffer class which can be found [here](#).

Buffer allows node.js developers to access data as it is in its internal representation (as per its memory allocation) and returns the number of bytes used in the particular case. On the contrary, String takes the encoding and returns the number of characters used in the data.

While working with binary data, developers frequently need to access data that have no encoding - like the data in image files. Examples also include, reading an image file from a TCP connection or reading a file from the local disk etc.

3.4 Event Emitter in Node.js

The Javascript classes which inherit from Event Emitter in Node.js, generally are a source of different event/events collection in our apps. With those classes we can listen in the `.on()` function on the object for the event that was specified in the arguments and work accordingly with the callback function codes.

Example:

```
var str = '';  
  
someObject.on('dataReceived', function(data) {  
    dataReceived += data;  
})  
.on('end', function() {  
    console.log('The data received: ' + data);  
})
```

The `on()` function returns a reference to the object it is attached to and can chain n number of event listeners.

Now if we have to emit events from a function we need to write the following code:

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

In order to have inheritance functionality with EventEmitter in Javascript, we have to inherit the particular class with:

```
util.inherits(myEventEmitterClass, EventEmitter);
```

Here the `myEventEmitterClass` will be the class which will inherit the EventEmitter functionality.

And then the actual Event emitting will happen in the following code:

```
myEventEmitterClass.prototype.emittedMethod = function() {  
    console.log('before the emittedMethod');  
    this.emit('emittedevent');  
    console.log('after the emittedMethod');  
}
```

So, while the `myEventEmitterClass` will be used in the system, the `emittedEvent` will be exposed like:

```
someObject.on('emittedevent', function() {  
    console.log('The emittedevent is called');  
})
```

The above is a short introduction of the EventEmitter in node.js. Utilizing this functionality can be a little bit tricky to a new node.js programmer, but will remain useful in various application areas.

3.5 Download the Source Code

This was a tutorial of Buffers and Modules in Node.js. You may download the source code of this tutorial: [Node_TestValidation.zip](#)

Chapter 4

Full application example

4.1 Introduction

As we have discussed, Node.js is a javascript runtime. Unlike traditional web servers, there is no separation between the web server and our code and we do not have to customize external configuration files (XML or Property Files) to get the Node.js Web Server up. With Node we can create a web server with minimal code and deliver content to the end users easily and efficiently. In this lesson we will discuss how to create a web server with Node and how to work with static and dynamic file contents, as well as about performance tuning in the Node.js Web server.

For this application, we will use the Node.js server for the web controller and the contents routing. We will also persist and fetch data through CSV files by using a package (`ya-csv`) through NPM (Node Package Manager Registry). Finally, we will perform front end rendering with EJS (Embedded JavaScript), which also resides within the node package manager `ejs` module.

Below is the architecture of the application in a nutshell:

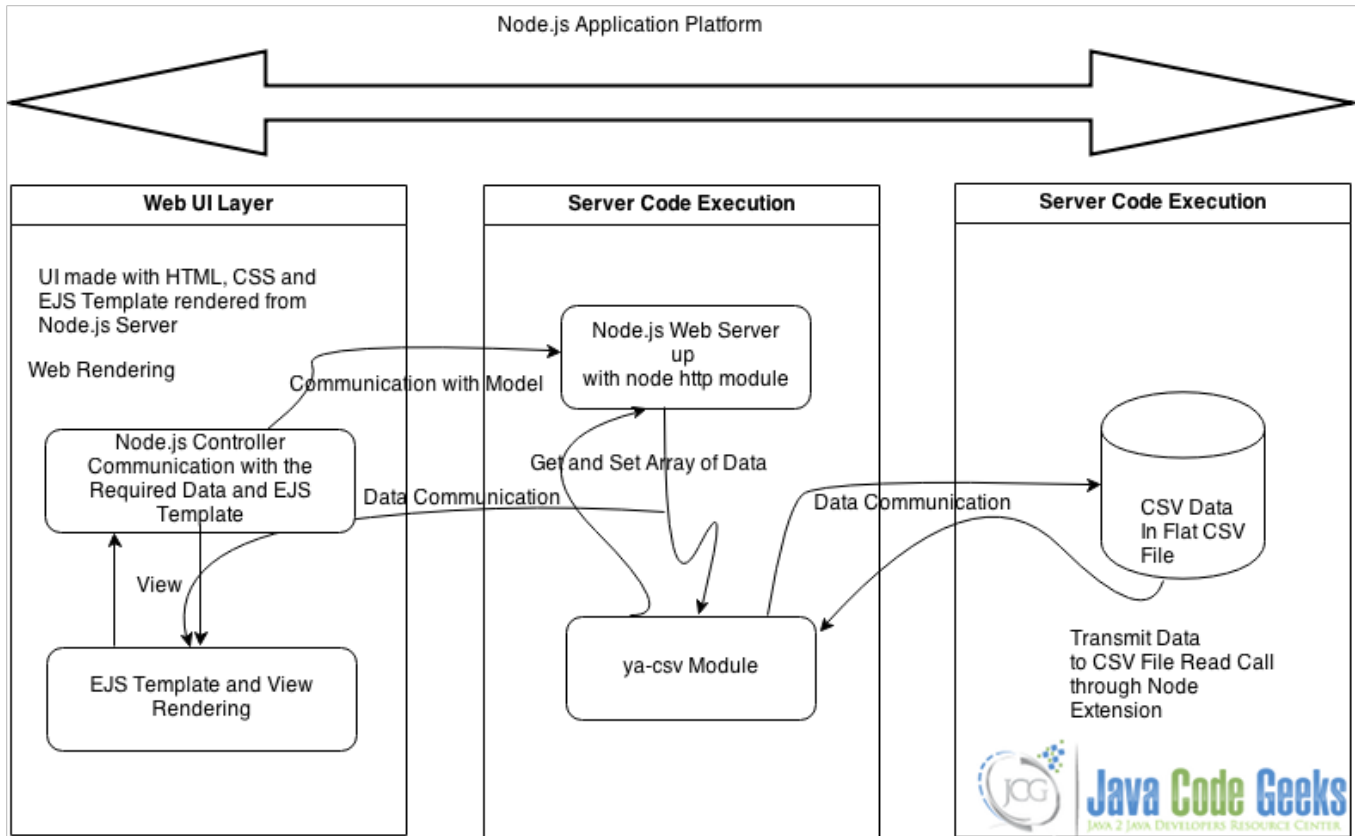


Figure 4.1: screenshot

Before going forward (assume we have completed the [node.js installation](#)), we need to issue the following command to have all the required libraries.

```
npm install ya-csv
npm install ejs
```

You can find the code here: [download](#).

4.2 Application Frontend rendering with EJS

Javascript templates are rendered and cached in the client-side without sending an HTTP request to the server. They are introduced as a layer in HTML in order to separate the HTML string appending from the server side. Additionally, they are used to get the required data from the server and then render this data according to the pre-defined templates of HTML Fragments.

So here comes EJS (Embedded Javascript). EJS provides flexibility for using JavaScript templates. Some of the EJS Templates are used in this Example Application. EJS is used by loading a template and rendering it with data from server side. Once a template for EJS is loaded, it is cached by default.

Example:

```
template = new EJS({url: '/firstejstemplate.ejs'})
```

We can provide extra options as arguments, which configure the following attributes:

url (string): Loads template from a file

text (string): Uses the provided text as the template

name (string) (optional): An optional name that is used for caching. This defaults to the element ID or the provided URL
cache (boolean) (optional): Defaults to true. If we want to disable caching of templates, we make this false

4.3 EJS Rendering

There are 2 ways to render a template.

- Invoke method *render*, which simply returns the text.
- Invoke method *update*, which is used to update the inner HTML of an element.

4.3.1 render (data) method

As we said, this method renders the template with the provided data.

```
html = new EJS({url: '/template.ejs'}).render(data)
```

In our example application we have not used the update method, as we stick to the basics and not provide any ajax functionality. For example, the code mentioned above is used in the file `registeredusers.ejs` (HTML Rendering):

```
<% for(var i=0; i<users.length; i++) { %>  
<%= users[i][0] %><%=users[i][1] %><%=users[i][2] %><%=users[i][3] %><%=users[i][4] %>  
<% } %>
```

This is the templating section where a table is rendered based on a javascript array. The array is pre-populated with the values in the Server.

So in `server.js` (data population section) we have:

- `var ejs = require('ejs');`: a variable declaration for ejs component.

Data population through server:

```
ejs.renderFile('content/registeredusers.ejs', { users : data },  
  function(err, result) {  
    // render on success  
    //console.log("this is result part -->" + result);  
    if (!err) {  
      response.end(result);  
    }  
    // render or error  
    else {  
      response.end('An error occurred');  
      console.log(err);  
    }  
  });
```

Here the `registeredusers.ejs` is a template file. And the `users` array is populated with some data array from previous section, which we will examine later. So the `renderFile` method will render the data with the template file through the standard callback function and asynchronous mode.

More on EJS can be found [here](#).

4.4 The Node.js Server

In the node.js platform, we start up the server as follows:

```
var http = require('http');
http.createServer(function (request, response) {
    ....
});
```

Let's now discuss the different features in server.js:

- We have used the path module to extract the basename of the path (the final part of the path) and reverse URI encoding from the client with the decodeURI method. We have stored all the static html, css, js and template files (EJS) in a folder named content.

```
var lookup = path.basename(decodeURI(request.url)) || 'index.html',
    f = 'content/' + lookup;
```

- We assigned the callback function to http.createServer which provides us access to the standard web request and response objects. We used the request object to find the requested URL and we get it's basename with *path*. We use the decodeURI method which will try to find out the page for the requested URL from the "content" folder. So with the basename we will match the resource as per our request.
- From here we have called additional functions to retrieve data, persist the data or match the filename and send it to for rendering with the loading contents. We have used the fs (filesystem) module to load the content and then return that content to the createServer callback.

```
// Data fetch from user form posting and persist
if (request.method === "POST") {
    ...Some Code
    request.on('data', function (chunk) {

        postData += chunk;
        .... Some Code
    }).on('end', function () {

        ....some code
    });
```

or

```
// Fetching the files and templates from filesystem and render the response to the client
if (request.method === "GET") {
    fs.exists(f, function (exists) { // Use of fileSystem Module

        ....some code
        // Serving the file in chunk for response -

        var s = fs.createReadStream(f).once('open', function () {
            console.log('stream deliver');
            response.writeHead(200, headers);
            this.pipe(response);
        }).once('error', function (e) {
            console.log(e);
            response.writeHead(500);
            response.end('Server Error!');
        });
        ....some code
```

```

    }
    // Response while no such content found
    response.writeHead(404);
    response.end('Page Not Found!');
  });
};

```

- Now for each of the client request, we should not open the static resources for each call from the client. So we should offer a minimum caching system for this. The application will serve the first request from the static storage, i.e. from the file/s and store it in cache. Afterwards it will serve all subsequent requests for static resources from the cache i.e. from the process memory.

So our bare-bones caching code will look like this:

```

var cacheObj = {
  cachestore: {},
  maxSize: 26214400, // (bytes) 25mb
  maxAge: 5400 * 1000, // (ms) 1 and a half hours
  cleaninterval: 7200 * 1000, // (ms) two hours
  cleanetimestart: 0, // to be set dynamically
  clean: function (now) {
    if ((now - this.cleaninterval) > this.cleanetimestart) {
      console.log('cleaning data...');
      this.cleanetimestart = now;
      var that = this;
      Object.keys(this.cachestore).forEach(function (file) {
        if (now > that.cachestore[file].timestamp + that.maxAge) {
          delete that.cachestore[file];
        }
      });
    }
  }
};

```

We have added the code to read the file just once and the load the contents onto the memory. After that, we will respond to client requests for the file from the memory. Also we have set a maxAge parameter, which is the time interval after which the cache gets cleaned up. We have also defined the maximum file size via the maxSize parameter.

Here is a code snippet with which we store the file in the cache store:

```

...some code....
fs.stat(f, function (err, stats) {
  if (stats.size < cacheObj.maxSize) {
    var bufferOffset = 0;
    cacheObj.cachestore[f] = {content: new Buffer(stats.size),
      timestamp: Date.now()};
    s.on('data', function (data) {
      data.copy(cacheObj.cachestore[f].content, bufferOffset);
      bufferOffset += data.length;
    });
  }
});
...some code....

```

Here is a code snippet with which we serve the client requests from the cache store:

```

fs.exists(f, function (exists) {
  if (exists) {
    var headers = {'Content-type': mimeTypes[path.extname(f)]};
    if (cacheObj.cachestore[f]) {

```

```
    console.log('cache deliver');
    response.writeHead(200, headers);
    response.end(cacheObj.cachestore[f].content);
    return;
  }
  ....other codes....
```

Here, the `fs` variable refers to the "fs" module of node.js and the `f` refers to file object.

In the above code, first we store the file in the `cachestore` array of cache object.

The next time any client requests the file, we check that we have the file stored in the cache object and will retrieve an object containing the cached data. In the `on` callback of the `fs.stat` method, we have made a call to `data.copy` in order to store the data in the `content` property of the `cachestore` array.

Please note that if we make any changes to the contents of the file, those will not be reflected to the next client, as those file copies are served from the cache memory. We will need to restart the server in order to provide the updated version of the content.

Caching content is certainly an improvement in comparison to reading a file from disk for every request.

4.5 Use of Streaming Functionality

For better performance, we should stream file from disk and then pipe it directly to the response object, sending data to the network socket one piece at a time. So we have used `fs.createReadStream` in order to initialize a stream, which is piped to the response object. In this case, the `fs.createReadStream` will need to interface with the request and the response objects. Those are handled within the `http.createServer` callback.

Here we have filled the `cacheObj[f].content` while interfacing with the `readStream` object.

The code for `fs.createReadStream` is the following:

```
var s = fs.createReadStream(f).once('open', function () {
  console.log('from stream');
  response.writeHead(200, headers);
  this.pipe(response);
}).once('error', function (e) {
  console.log(e);
  response.writeHead(500);
  response.end('Server Error!');
});
```

This will return a `readStream` object which streams the file that is pointed at by the `f` variable. `readStream` emits events that we need to listen to. We listen with the shorthand `once`:

```
var s = fs.createReadStream(f).once('open', function () {
  //do stuff when the readStream opens
});
```

Here, for each file, each stream is only going to open once and we do not need to keep listening to it.

On top of that we have implemented error handling with the following piece of code:

```
var s = fs.createReadStream(f).once('open', function () {
  //do work when the readStream opens
}).once('error', function (e) {
  console.log(e);
  response.writeHead(500);
  response.end('Server Error!');
});
```

Now the `stream.pipe` or `this.pipe` method enables us to take our file from the disk and stream it directly to the socket through the standard response object. The `pipe` method detects when the stream has ended and calls `response.end` for us.

We had used the data event to capture the buffer as it's being streamed, and copied it into a buffer that we supplied to `cacheObj[f].content`. We achieved that by using `fs.stat` to obtain the file size for the file's cache buffer.

What we achieved is that instead of the client waiting for the server to load the full file from the disk before sending it back, we used the stream to load the file in small one-by-one pieces. Especially for larger files this approach is very useful, since there will be a delay time between the file being requested and the client starting to receive the file.

We did this by using `fs.createReadStream` to start the streaming of the file from the disk. Here the method `fs.createReadStream` creates the `readStream`, which inherits from the `EventEmitter` class. We had used listeners like `on` to control the flow of stream logic:

```
s.on('data', function (data) {
    ...code related to data processing...
});
```

Then, we added an open event listener using the `once` method as we want the call for open once it has been triggered. We respond to the open event by writing the headers and using the `pipe` method to send the data to the client. Here `stream.pipe` handles the data flow.

We also created a buffer with a size (or an array or string) which is the size of the file. To get the size, we used the asynchronous `fs.stat` and captured the size property in the callback. The data event returns a `Buffer` as the callback parameter.

Here a file with less than the `bufferSize` will trigger one data event because the entire file will fit into the first chunk of data. For files greater than `bufferSize`, we will fill the `cacheObj[f].content` property one piece at a time.

We managed to copy little binary buffer chunks into the binary `cacheObj[f].content` buffer. Each time a new chunk comes and gets added to the `cacheObj[f].content` buffer, we update the `bufferOffset` variable by adding the length of the chunk buffer to it. When we call the `Buffer.copy` method on the chunk buffer, we pass the `bufferOffset` as the second parameter to fill the buffer correctly.

Additionally, we have created a `timestamp` property into `cacheObj`. This is for cleaning up the cache, which can be used as an alternative to restarting the server after any future content change.

The `clean` method in the `cache` class is as follows:

```
clean: function(now) {
    Object.keys(this.store).forEach(function (file) {
        if (now > that.cachestore[file].timestamp + that.maxAge) {
            delete that.cachestore[file];
        }
    });
}
```

Here we have checked the `maxAge` property for all the objects in the cache store and simply deleted the object after `maxAge` expires for an object. We have called the `cache.clean` at the bottom of the server like this: `cacheObj.clean(Date.now());`

`cacheObj.clean` loops through `cache.store` and checks to see if the cached objects have exceeded their specified lifetime. We set the `maxAge` parameter as a constant for the cleanup time. If a cache objects has expired, we remove it from the store.

We had added the `cleaninterval` to specify how long to wait between cache cleans. The `cleanetimestamp` is used to determine how long it has been since the cache was last cleaned.

The code for that is the following:

```
cleaninterval: 7200 * 1000, // (ms) two hours
cleanetimestamp: 0, // to be set dynamically
```

The check in the `clean` method is as follows:

```
if (now - this.cleaninterval > this.cleanetimestamp) {
    // Do the work
}
```

Here the clean will be called and clean the cache store only if it has been longer than `cleaninterval` variable value after the last clean.

The above code was a simple example of building an minimal `cacheObj` object within the node.js server.

4.5.1 The model part - csv handling

While we are getting the data from the post method, we are opening a csv file in append mode. Then we have written the data from post parameters to the csv file. After that, we have read the data and render the data with EJS template and send it to the client.

Below is the code snippet along with the relevant documentation:

```

if (request.method === "POST") { // Checking the request method.
    var postData = '';
    request.on('data', function (chunk) {

        postData += chunk; // accumulate the data with the 'chunk'
        console.log('postData.length-->' + postData.length);

        if (postData.length > maxData) { // checking for a max size of data
            postData = '';
            this.pause();
            response.writeHead(413); // Request Entity Too Large
            response.end('Too large');
        }

    }).on('end', function () { // listener for the signal of end of data.
        if (!postData) { response.end(); return; } //prevents empty post requests ←
        // from crashing the server
        var postDataObject = querystring.parse(postData);
        var writeCSVFile = csv.createCsvFileWriter('users.csv', {'flags': ' ←
        a'});
        // opening the file in append mode
        var data = [postDataObject.name, postDataObject.email, postDataObject ←
        .location, postDataObject.mobileneno, postDataObject.notes]; // ←
        // preparation of the data array

        writeCSVFile.writeRecord(data); // Writing to the CSV file
        console.log('User Posted 1234:\n', JSON.stringify(util.inspect( ←
        postDataObject)));

        var readerCSV = csv.createCsvFileReader('users.csv'); // Opening of ←
        // the file
        var data = [];
        readerCSV.on('data', function(rec) {
            data.push(rec); // getting all data in memory
        }).on('end', function() {
            console.log(data);
            ejs.renderFile('content/registeredusers.ejs', { users : data ←
            }, // Rendering of the data with corresponding EJS ←
            // template
            function(err, result) {
                // render on success
                //console.log("this is result part -->" + result);
                if (!err) {
                    response.end(result);
                }
                // render or error
                else {
                    response.end('An error occurred');
                    console.log(err);
                }
            }
        });
    });
}

```

```
    });  
  });  
});
```

To sum up, in this article, we have discussed about:

- The Node.js web server creation.
- A minimal cache object creation for data handling.
- Streaming of data for optimised performance.
- Handling data from a GET request.
- Handling data from a POST request.
- A dynamic router for controlling navigation.

There is certainly room for improvement of the node.js code. For example, we could add separation of the CSV handling code in a separate model layer, something that would make the design more elegant, and functional separation of the Node Middleware components, like the caching and independent model handling. These enhancements are left to the reader as exercise.

4.6 Download the Source Code

In this tutorial we discussed building a full application in Node.js. You may download the source code here: [Nodejs_full-app.zip](#)

Chapter 5

Express tutorial

5.1 Introduction

Express.js is a powerful web development framework for the Node.js (Node) platform. Express.js comes with the rest of the Node.js middleware modules. These modules are JavaScript components which can be used in Node.js based web applications to make the application layered and more modular.

With express.js, except for the express.js APIs, other node.js core APIs can also be called. The Express.js framework can be used in order to develop any kind of web application, from simple to complex ones. As always, when developing with Node.js, we have to keep in mind the asynchronous behaviour of our application.

5.2 Express.js Installation

After **installing Node.js** in our machine, it is easy to also install express.js. This is accomplished via the node.js package manager. We need to install express.js as a node module with the `-g` option as an npm install command at terminal.

The command to install Express is the following:

```
$ sudo npm install express -g
```

This will install the latest stable version of Express.

If we need to install express.js locally, then we need to run the following:

```
$ sudo npm install express
```

The difference between local and global installation is this: for local installation, the express.js will be available in the `node_modules` folder within the express.js folder, while for the global installation, the express.js will be available at the node executable path for all the applications which are developed on the particular machine.

The express.js installation can be confirmed by requesting the version for express.js. Run the following in the terminal:

```
$ express -V  
> 3.4.7
```

5.3 Express.js Objects

5.3.1 The application object

The application object is an instance of Express, generally presented by the variable named `app`. This is the main object of our Express application and all functionality is built on top of that object.

Creating an instance of the Express.js module within the node application is done as follows:

```
var express = require('express');
var app = new express();
```

5.3.2 The request object

Now, when a web client makes a request to the Express application, the HTTP request object is created. Inside all the application callbacks, whenever the request objects are passed as reference, those are represented with the conventional variable `req`. This request object holds all the HTTP stack related variables, such as headers information, HTTP methods and related properties for that particular request from the web client.

Let's see the most important methods of Request object:

- `req.params`: Holds the values of all the parameters of the request object.
- `req.params(name)`: Returns value of a specific parameter from the Web URL GET params or POST params.
- `req.query`: Takes values from a GET method submission.
- `req.body`: Takes values from a POST form submission.
- `req.get(header)`: Gets the request HTTP header.
- `req.path`: The request path.
- `req.url`: The request path with query parameters.

5.3.3 The response object

The response object is created along with the request object, and is generally represented by a variable named `res`. In the HTTP Request-Response model, all the express middlewares work on the request and the response objects while one is passing the control after another.

Let's see the most important methods of Response object:

- `res.status(code)`: The HTTP response code.
- `res.attachment([filename])`: The response HTTP header Content-Disposition to attachment.
- `res.sendFile(path, [options], Sends a file to the client [callback])`.
- `res.download(path, [filename], Prompts the client to download from [callback])`.
- `res.render(view, [locals], Renders a view callback)`.

5.4 Concepts used in Express

5.4.1 Asynchronous JavaScript

Node.js programming is mainly based on Asynchronous Javascript Programming. All of the modules in node.js are built asynchronous in nature. So, the execution of the code from one layer to another generally is performed within callback functions.

As node.js program executes in an event loop, the end user generally does not face any "blocking" from the view layer, i.e. the web browser or mobile browser etc. Generally the callback function is passed as an async function to be executed. This function will return the result to the upper function when the execution of code is completed.

Note that all the programs within express.js and the associated programs are installed in the node.js environment as node modules. For any node.js application, the deployment configuration is written in a "package.json" file. If we need to install the application as a node module in the node.js environment, i.e. through the npm install command, we should also include the package.json file.

5.4.2 Middlewares in node.js applications

A middleware in a node.js application context is a JavaScript function which will handle HTTP requests. It will be able to handle the request and the response objects from HTTP request, perform some operation on the request, send back the response to the client and finally pass the objects/results to the next middleware. Middlewares are loaded in an Express application with the `app.use()` method.

A basic example of a middleware is one that will handle a GET request method, as follows:

```
app.use(function(req, res, next) {
  console.log('Request Query : ' + req.query);
  next();
});
```

This will print the "querystring" within the request object.

The majority of the Express.js functionality is implemented with its built-in middlewares. One of the Express.js middleware is the router middleware, which is responsible for routing the HTTP requests inside Express applications to the appropriate data handler functions. From a user perspective, it is the navigational functionality in a web application.

The destinations of the HTTP request URIs are defined via routes in the application. Routes are the controlling points for the response from a request, i.e. they decide where to dispatch a specific request by analysing data carried on the request object. In traditional web applications, like a JEE Application, this functionality is handled by the Controller in the application. Route handlers may be defined in the `app.js` file or loaded as Node modules.

Defining the routes and their handlers in the `app.js` file works fine when the number of routes is relatively small.

The following code presents an example of `routes.js` in the Node module:

```
module.exports = function(app) {
  app.get('/', function(req, res) {
    // Send a plain text response
    res.send('First Express Application!');
  });

  app.get('/hello.text', function(req, res) {
    // Send a plain text response
    res.send('Hello!');
  });

  app.get('/contact', function(req, res) {
    // Render a view named contact
    res.render('contact');
  });
};
```

The `app.js` file can be defined as below:

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

A request handler can send a response back to the client using response methods in the response object.

5.5 Definition of Routes

Generally, a URL (Uniform Resource Locator) combines the HTTP request method (GET or POST) and the path pattern for a web application. The routes handle the URL with the appropriate route handler, which executes the job for any particular action

for that URL. A request to the server that matches a route definition is routed to the associated route handler. The route handlers are also able to send the HTTP response or pass the request to the next middleware, if necessary.

Routes in Express are defined using methods named as the HTTP methods, for example `app.get()`, `app.post()`, `app.put()` and so on.

There is also another set of string-based route identifiers, which are used in order to specify placeholders for the request path.

The following example demonstrates this concept:

```
app.get('/user/:id', function(req, res) {
    res.send('user id: ' + req.params.id);
});
app.get('/country/:country/state/:state/city/:city', function(req, res) {
    res.send(req.params.country + ', ' + req.params.state + ', ' + req.params.city);
}
```

So, the value of the placeholder name will be available in the `req.params` object which we can access as in the above example.

Below is a code snippet for an optional parameter:

```
app.get('/feed/:datatype?', function(req, res) {
    if (req.params.datatype) { res.send('Data Type: ' + req.params.datatype); }
    else { res.send('Text Data'); }
});
```

5.5.1 How to handle routes in express.js

When a request is made to the server, for the given route definition, the associated callback function will be executed in order to process the request and send back the response. These callback functions generally define the behaviour of our application.

5.5.2 Namespaced Routing

We can define the routes on the basis of a namespace, which will act as the root path and then the rest of the routes will be defined relatively to that route. By default, `express.js` does not have the capability for namespaced routing. For getting the advantage of this functionality, we will have to install the `express-namespace` node module.

The command for the installation is the following:

```
$ npm install express-namespace
```

So now we can write the `app.js` as follows:

```
var http = require('http');
var express = require('express');
// express-namespace should be loaded before app is instantiated
var namespace = require('express-namespace');
var app = express();

app.use(app.router);
app.namespace('/posts', function() {
    app.get('/', function(req, res) {
        res.send('all posts');
    });
    app.get('/new', function(req, res) {
        res.send('new post');
    });
    app.get('/edit/:id', function(req, res) {
        res.send('edit post ' + req.params.id);
    });
    app.get('/delete/:id', function(req, res) {
```

```
        res.send('delete post ' + req.params.id);
    });
    app.get('/2013', function(req, res) {
        res.send('articles from 2014');
    });
    // Namespaces can be nested
    app.namespace('/2014/jan', function() {
        app.get('/', function(req, res) {
            res.send('posts from jan 2014');
        });
        app.get('/angularjs', function(req, res) {
            res.send('articles about Angular.js from jan 2014');
        });
    });
});
http.createServer(app).listen(3000, function() {
    console.log('App started');
});
```

We can see the results for the following urls:

```
http://localhost:3000/posts/
http://localhost:3000/posts/edit/1
http://localhost:3000/posts/delete/1
http://localhost:3000/posts/2014
http://localhost:3000/posts/2014/jan
http://localhost:3000/posts/2014/jan/angularjs
```

5.6 HTTP response in Express

A response from express.js can be generated with a minimal route and a callback as below:

```
app.get('/', function(req, res) {
    res.send('our application response');
});
```

Express.js can handle any type of Error in Application which are standard HTTP response error codes.

5.6.1 Setting the HTTP status code

We can set the HTTP status code by passing the number to the `res.status()` method.

An example for sending the 404 Status code can be found below:

```
app.get('/', function(req, res) {
    // Set the status
    res.status(404);
    // Specify the body
    res.send('404 Error');
});
```

By default, express.js sends response code 200.

Also we can send the status as a chained response:

```
app.get('/', function(req, res) {
    // Status and body in one line
    res.status(404).send('resource not found');
});
```

Some HTTP methods, like `res.send()`, `res.json()`, and `res.jsonp()` are capable of sending the HTTP status code themselves.

In case of sending a 200 status code:

The example is for `res.send()`:

```
app.get('/', function(req, res) {
  res.send('welcome');
});
```

Now if a number is passed in the body, it will be like:[source.java]

```
app.get('/', function(req, res) {
  res.send(404);
});
```

or

```
app.get('/', function(req, res) {
  res.send(404, 'not found');
});
```

5.6.2 Setting HTTP headers

Express provides an interface for setting HTTP headers in the response message. We have to pass two parameters to the `res.set()` method; the first parameter is the header name and the second parameter is the value of the parameter.

An example for setting the standard HTTP header with custom header variables is the following:

```
app.get('/', function(req, res) {
  res.status(200);
  res.set('Content-Type', 'text/plain; charset=us-ascii');
  res.set('X-Custom-Message', 'it is a custom message');
  res.send('HTTP header setting example');
});
```

5.6.3 Sending data

If we want to serve plain text with the response, we can write:

```
app.get('/', function(req, res) {
  res.send('</pre>

<h2>Plain text</h2>

');
});
```

It will show `<h1>Plain text</h1>` in browser.

Equivalent to this will be:

```
app.get('/', function(req, res) {
  res.set('Content-Type', 'text/plain');
  res.send('</pre>

<h2>Plain text</h2>

<pre>
');
});
```

If we want to see the HTML Equivalent of this, then we should try:

```
app.get('/', function(req, res) {
  res.set('Content-Type', 'text/html');
  res.send('</pre>

<h2>Plain text</h2>
<pre>
');
});
```

If we want json as output, we can do this by setting the json object in response i.e. by using the res.json method.

```
app.get('/', function(req, res) {
  res.json({message: 'This is example of json'});
});
```

It will set a default 200 HTTP Status with the response.

For responding with content negotiation, the response object will return like the following with res.format method:

```
app.get('/', function(req, res) {
  res.format({
    'text/plain': function() {
      res.send('Plain text');
    },
    'text/html': function() {
      res.send('*welcome*');
    },
    'application/json': function() {
      res.json({ message: 'welcome' });
    },
    'default': function() {
      res.send(406, 'Not Acceptable');
    }
  });
});
```

The server will respond based on the data type mentioned in the HTTP Accept header.

Now we will try to make a complete node.js application where the GET and POST methods are used from Express.js and will see how all these are tied together.

5.7 Sample web application with NeDB

We have chosen nedb for our datastore to make the application simple for reader. NeDB is a javascript based Embedded database. According to [this](#), NeDB is an "Embedded persistent database for Node.js, written in Javascript, with no dependency (except npm modules of course)."

To make the application, we have selected:

- Angular.js for client side development - Single Page Application.
- Cross Domain Communication in between Angular.js and Node.js.
- Node.js for server side development.
- Rest based web service creation with express.js.
- Database - NeDb.
- Node.js NeDB Module Extention.

We have created a Proof of Concept with a Javascript based web server, where we utilized NeDB with the javascript based framework Node.js and angular.js on the client side.

The architecture at a glance:

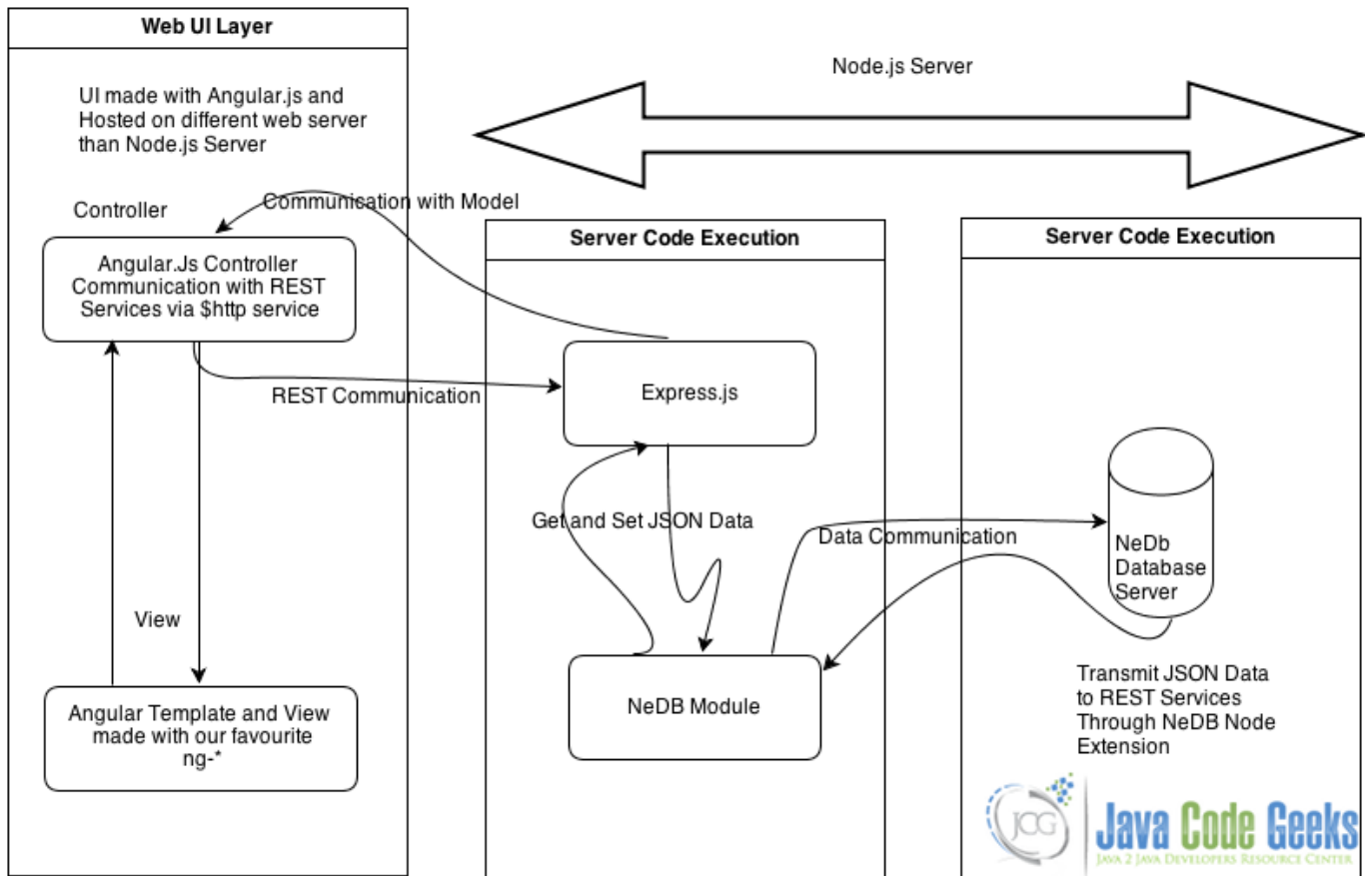


Figure 5.1: screenshot

Here are the steps:

5.7.1 Installation

- Download and install Node.js as described [here](#).
- To Develop the application we need to install nedb module for Node.js.

```
npm install nedb
```

- We need to install express.js for node.js.

```
npm install express
```


5.7.2 Configuration Code

Now, we will try to describe the used code portion:

```
var application_root = __dirname,
    express = require("express"),
    path = require("path");
```

Here we have initialised the express.js based on the Node.js concepts discussed above.

```
var app = express();
```

Here we have initialised the express web server and reference the app variable.

```
var databaseUrl = "/home/sampleuser/nedb/user.db";
var Datastore = require('nedb');
db = {};
db.users = new Datastore({ filename: databaseUrl, autoload: true }); // to autoload ←
    datastore
```

Here we have made the connection to the nedb database using the Node.js nedb module extension library.

```
// Config

app.configure(function () {
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
  app.use(express.static(path.join(application_root, "public")));
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true }));
});
```

Here we have made the configuration related to express.js.

5.7.3 Router Handling Code

```
app.get('/api', function (req, res) {
  res.send('Our Sample API is up...');
});
```

Here we have made our first REST based web service and tested whether the express.js is up.

Our sample api will be: <http://127.0.0.1:1212/api> (GET Method).

```
app.get('/getangularusers', function (req, res) {
  res.header("Access-Control-Allow-Origin", "http://localhost");
  res.header("Access-Control-Allow-Methods", "GET, POST");
  // The above 2 lines are required for Cross Domain Communication(Allowing the ←
    methods that will come as Cross Domain Request
  // More on this in later part of the code
  db.users.find('', function(err, users) { // Query in NeDB via NeDB Module
    if( err || !users) console.log("No users found");
    else
      {
        res.writeHead(200, {'Content-Type': 'application/json'}); // Sending data ←
          via json
        str='[';
        users.forEach( function(user) {
          str = str + '{ "name" : "' + user.username + '", ' + '\n';
        });
        str = str.trim();
```

```

        str = str.substring(0, str.length-1);
        str = str + ']';
        res.end( str);
        // Prepared the jSon Array here
    }
});
});

```

Here we have created another REST API to get all usernames from a user collection and so we have performed the necessary NeDB query.

Our sample api will be: <http://127.0.0.1:1212/getangularusers> (GET Method).

```

app.post('/insertangularneuser', function (req, res){
  console.log("POST: ");
  res.header("Access-Control-Allow-Origin", "http://localhost");
  res.header("Access-Control-Allow-Methods", "GET, POST");
  // The above 2 lines are required for Cross Domain Communication(Allowing the methods ↔
  // that come as Cross
  // Domain Request
  console.log(req.body);
  console.log(req.body.mydata);
  var jsonData = JSON.parse(req.body.mydata);

  db.users.save({email: jsonData.email, password: jsonData.password, username: jsonData. ↔
  username},
    function(err, saved) { // Query in NeDB via NeDB Module
      if( err || !saved ) res.end( "User not saved");
      else res.end( "User saved");
    });
});
});

```

Here we have made a POST request to create a user via a REST invocation.

Our sample api will be: <http://127.0.0.1:1212/insertangularneuser> (Post Method).

```

// Launch server
app.listen(1212);

```

We have made the server to listen at 1212 port.

Now we can run node appnedbangular.js from command prompt/terminal.

5.7.4 Angular.js part

We have used Angular.js for our client side work and development. We have made the choice of Angular.js as our front-end development tool as this maintains a clear client-side Model-View-Presenter Architecture and makes the code more structured.

Since this part of the tutorial mainly concentrates on node.js and express.js, the reader is urged to acquire more knowledge about Angular.js [here](#).

We provide inline documentation with our code to help you better understand the application with Angular.js

So, below is the code for the Angular Controller.

```

'use strict';

var myApp = angular.module('myApp', []); // Taking Angular Application in Javascript ↔
Variable

// Below is the code to allow cross domain request from web server through angular.js
myApp.config(['$httpProvider', function($httpProvider) {
  $httpProvider.defaults.useXDomain = true;

```

```
        delete $httpProvider.defaults.headers.common['X-Requested-With'];
    }
});

/* Controllers */

function UserListCtrl($scope, $http, $templateCache) {

    var method = 'POST';
    var inserturl = 'http://localhost:1212/insertangularneuser'; // URL where the Node.js server is running
    $scope.codeStatus = "";
    $scope.save = function() {
        // Preparing the Json Data from the Angular Model to send in the Server.
        var formData = {
            'username' : this.username,
            'password' : this.password,
            'email' : this.email
        };

        this.username = '';
        this.password = '';
        this.email = '';

        var jdata = 'mydata='+JSON.stringify(formData); // The data is to be string.

        $http({ // Accessing the Angular $http Service to send data via REST Communication to Node Server.
            method: method,
            url: inserturl,
            data: jdata ,
            headers: {'Content-Type': 'application/x-www-form-urlencoded'},
            cache: $templateCache
        }).
        success(function(response) {
            console.log("success"); // Getting Success Response in Callback
            $scope.codeStatus = response.data;
            console.log($scope.codeStatus);

        }).
        error(function(response) {
            console.log("error"); // Getting Error Response in Callback
            $scope.codeStatus = response || "Request failed";
            console.log($scope.codeStatus);

        });
        $scope.list();// Calling the list function in Angular Controller to show all current data in HTML
        return false;
    };

    $scope.list = function() {
        var url = 'http://localhost:1212/getangularusers'; // URL where the Node.js server is running
        $http.get(url).success(function(data) {
            $scope.users = data;
        });
        // Accessing the Angular $http Service to get data via REST Communication from Node Server
    };

    $scope.list();
}
```

```
}

```

5.7.5 Angular Template and HTML

```
<html lang="en" ng-app="myApp">
.....

```

We refer to the Angular Application in above code.

```
<body ng-controller="UserListCtrl">
.....

```

We refer to the Angular Controller in above code.

Search:

```
<input ng-model="user">

  <!--Body content-->
  <ul class="users">
    <li ng-repeat="user in users | filter:user ">
      {{user.name}}
  </ul>

```

We have used the ng-repeat tag to take the users data model from the REST invocation.

```
<form name="myform" id="myform1" ng-submit="save()">
  <fieldset>
    <legend>New User</legend>

    <center><input type="text" placeholder="User..." ng-model="username" size=
    =50 required/></center>
    <center><input type="text" placeholder="Password..." ng-model="password"
    size=50 required/></center>
    <center><input type="text" placeholder="Email..." ng-model="email" size=50
    required/></center>

  </fieldset>
  <p>
    <center><button type="submit" >Save now...</button></center>
  </p>
</form>

```

We have used the ng-submit tag to send the user data model from the REST invocation and send to the node server in order to persist it in the NeDB database.

Please note that NeDB is just a light-weight Database which can be embedded in Node WebKit Applications. For fairly large scale database production systems, we should consider MongoDB.

5.8 Download the Source Code

This tutorial discussed Express.js and how to use it along with Node.js. You may download the source code here: [Node.js-NeDB.zip](#)

Chapter 6

Command line programming

6.1 Introduction

Node.js is one of the most hyped technologies over the past few years. It is built with JavaScript and runs on the Chrome V8 engine Runtime.

Node.js offers the following features while developing Applications:

- Develop once, run everywhere, from command line, server, or the browser.
- Event-driven programming.
- Non-blocking Input-Output Model.
- The main event loop from Node.js runs in Single thread.

In this article, we will discuss about node.js command line programming. We will create a sample program which processes any file and interacts with users.

6.2 Utility Programs with Node.js

Some useful modules are available with node.js, such as:

- Processing some data and preparing reports with node.js.
- Getting twitter data and storing them in the repository for further processing.

So, first let's build a simple program on to show you how to process command line arguments in node.js

Our Program listing is nodecmd.js.

nodecmd.js:

```
process.argv.forEach(function (val, index, array) {  
    console.log(index + ': ' + val);  
});
```

Once we run this program with:

```
node nodecmd.js one two three
```

we will have the output as follows:

```
0: node
1: /nodeapps/nodecommandline/nodecmd.js
2: one
3: two
4: three
```

Here, the first printing line is node - the program. Second one is the program file name. The other three lines are printouts of the arguments.

`process` is a core module that comes with the node.js installation. More documentation on `process` can be found <http://nodejs.org/docs/latest/api/process.html#process> [here].

We will use a new module, named `commander.js`, for the command line processing in node.js> It has several useful options which we will look here.

First, in order to install `commander.js`, we issue the following command in the node.js console:

```
npm install commander
```

Below is the listing for the sample command line program:

`nodecommand.js`:

```
var program = require('commander');

program
  .version('0.0.1')

program
  .command('show [name]')
  .description('initialize command')
  .action(function(name) {
    console.log('Yes ' + name + '...I have started...');
  });

program
  .command('bye')
  .description('by command')
  .action(function() {
    console.log('Bye for now');
  });

program
  .command('*')
  .action(function(env) {
    console.log('Please enter a Valid command');
  });

program.parse(process.argv);
```

Now, some commands that we can run in the node.js console are:

1.

```
node nodecommand.js -V
```

It will show the version number for the program, which we have defined with

```
program
  .version('0.0.1')
```

2.

```
node nodecommand.js -h
```

It will show a utility console with all the commands and usages like:

- Usage: nodecommand.js [options] [command]
- Commands:
 - show [name]: initialize command
 - bye: by command
 - *
- Options:
 - -h, --help: output usage information
 - -V, --version: output the version number
 - 1. [source,perl]

```
node nodecommand.js show Piyas
```

It will output the following in the console:

```
Yes Piyas...I have started...
```

Here `Piyas` is passed as an argument in the `nodecommand.js` program.

More on the use of commander can be found [here](#).

Now we can also run the program without prior initialization with the node command.

We can save the above program content with the following line in the upper section of the code and save the file as `nodesamplecommand`:

```
#!/usr/bin/env node
```

Now if we run the statements:

```
./nodesamplecommand -h
```

or

```
./nodesamplecommand show Piyas
```

they will give the same output as the previous program.

The only thing we have to make sure is to have the program `node` in the system environment path.

6.3 Command Line and User Interaction

Let's continue with a simple user interaction program with the `node process` as below:

`nodeuserinteraction`:

```
#!/usr/bin/env node

process.stdin.resume();
process.stdin.setEncoding('utf8');
var util = require('util');

process.stdin.on('data', function (text) {
  console.log('received data:', util.inspect(text));
  if (text === 'exit\n') {
    complete();
  }
  else
  {
    invalidCommand();
  }
});

function complete() {
  console.log('There is nothing to do now.');
```

```
process.exit();
}

function invalidCommand() {
  console.log('Please enter the valid command');
```

```
}
```

In the execution of the program above, the `process` module will resume in the `node.js` console till some user input arrives. As long as the user enters some data except `exit`, the program will prompt as follows:

```
Please enter the valid command
```

The program will stop when we issue the `exit` command.

6.4 File Handling in Node.js Command Line Program

Now we will work with a sample program, which will work with files. The user will enter a file as input in the console. The program will extract the keywords from the file and write the words in a different file.

To get the keywords from the file, we will use module *keyword-extractor*:

```
npm install keyword-extractor
```

This will install the keywords extractor library in the local repository.

Now the program listing of `noderemovestopwords`:

```
#!/usr/bin/env node // Node environment path in Linux

var keyword_extractor = require("keyword-extractor");
fs = require('fs');

var program = require('commander');

program
  .version('0.0.1')

program
  .command('process [filename]')
  .description('initialize command')
  .action(function(filename) {
```



```

console.log(filename);
fs.readFile(fs.realpathSync(filename), 'utf8', function (err,data) {
  if (err) {
    return console.log(err);
  }
  //Keyword Extraction
  var extraction_result = keyword_extractor.extract(data,{
                                                                    language:"english",
                                                                    return_changed_case ←
                                                                    :true
                                                                    });

  fs.writeFile(filename+'.out.txt', extraction_result, function (err) {
    if (err) return console.log(err);
    console.log('File writing done');
  });
});
});
program.parse(process.argv);

```

To run the program, we need to write:

```
./noderemovestopwords process <>
```

which will output a file named <<File Name>>.out.txt in the output directory.

Let's now discuss the program:

- First we have taken the argument as `filename` in Command line.
- After this, we have read the file and processed the file contents using the keyword extraction library. As this keyword extraction process is a synchronous process, we have written the extracted content to a different file with the Node.js filesystem `writeFile` methods.
- And finally we have logged a message when the process ends within the `writeFile` callback function.

So this is an example of file handling through node.js command line program.

6.5 Publish the Program to NPM

NPM makes it really easy to share Node.js programs to the world. The Steps are:

- Create a configuration file for program.
- Execute the command `publish`.

There should be a `package.json` file for the configuration:

```

{
  "author": "Piyas De",
  "name": "noderemovestopwords",
  "url" : "http://www.phloxblog.in",
  "description": "A sample CLi program created to extract the keywords from a given file ←
    and output the keywords to a different file",
  "version": "0.0.4",
  "repository": {
    "type": "git",
    "url": "https://github.com/piyasde/nodecommandline"
  }
}

```

```
  },
  "main": "./bin/noderemovestopwords",
  "keywords": [
    "keyword",
    "extractor",
    "commandline"
  ],
  "bin": {
    "noderemovestopwords": "./bin/noderemovestopwords"
  },
  "dependencies": {
    "commander": "2.1.x",
    "keyword-extractor": "0.0.7"
  },
  "engine": "node >= 0.10.22",
  "license": "BSD"
}
```

A Note for the Folder Structure:

```
nodecommandline
|--src
  |--bin
  |   |--noderemovestopwords
  |--package.json
  |--README.md
```

Now we need to execute:

- `npm adduser`: to add the user name for whom the node.js package is uploaded.
- `npm publish`: to publish the node.js package to the Repository.

6.6 Download the Source Code

In this tutorial we discussed command line programming in Node.js. You may download the source code here: [Nodejs-Command_line.zip](#)