

Bruno Joseph Dmello

# What You Need To Know About Node.js

Dive into the fundamentals of Node.js



Packt>

# What You Need To Know About Node.js

Dive into the fundamentals of Node.js

**Bruno Joseph Dmello**



BIRMINGHAM - MUMBAI

# What You Need To Know About Node.js

Copyright © 2016 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First Published: November 2016

Production reference: 1241116

Published by Packt Publishing Ltd.

Livery Place

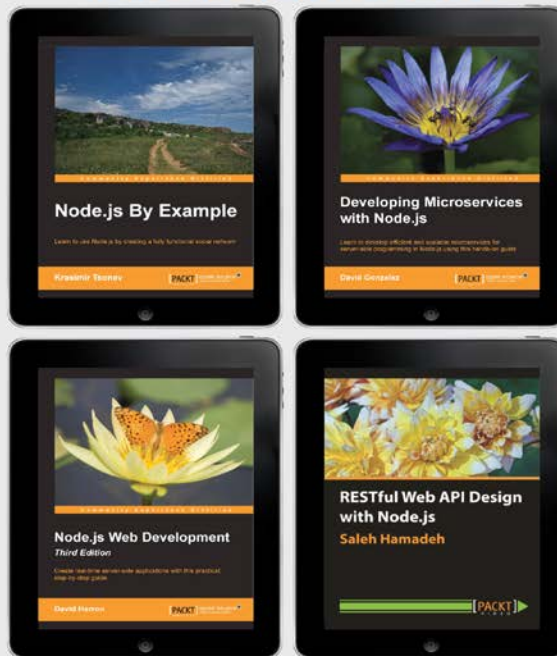
35 Livery Street

Birmingham B3 2PB, UK.

[www.packtpub.com](http://www.packtpub.com)

Get  
**50%**  
Off

Your next eBook or Video



Use the following code to  
apply your exclusive discount

**Node50**

# About the Author

**Bruno Joseph Dmello** is proactively working at Yapsody as a software engineer. He is a JavaScript enthusiast and loves working with open source communities. He possesses 4 years of experience in software development, of which 2.5 years were dedicated to working on JavaScript technologies. Bruno follows Kaizen and enjoys the freedom of architecting new things at work. He is socially active as well; he's involved in coaching technology stuff to freshers and participates in other projects, campaigns, and meet-ups.

You can connect with Bruno at [bron1010@gmail.com](mailto:bron1010@gmail.com).

---

Lastly I would like to acknowledge my family for their support and patience, especially Erina.

---

# About the Reviewer

**Alex Libby** has a background in IT support. He has been involved in supporting end users for almost 20 years in a variety of environments; a recent change in role now sees Alex working as an MVT test developer for a global distributor based in the UK. Although Alex gets to play with different technologies in his day job, his first true love has always been with the open source movement and, in particular, experimenting with CSS/CSS3, jQuery, and HTML5. To date, Alex has written 11 books on subjects such as jQuery, HTML5 Video, SASS, and CSS for Packt Publishing, and has reviewed several more. *Responsive Web Design with HTML5 and CSS3 Essentials* is Alex's twelfth book for Packt Publishing and his second book completed as a collaboration project.

# www.PacktPub.com

For support files and downloads related to your book, please visit [www.PacktPub.com](http://www.PacktPub.com). Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.PacktPub.com](http://www.PacktPub.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [service@packtpub.com](mailto:service@packtpub.com) for more details.

At [www.PacktPub.com](http://www.PacktPub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www.packtpub.com/mapt>

## Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser.

# What You Need To Know About Node.js

This eGuide is designed to act as a brief, practical introduction to Node.js. It is full of practical examples that will get you up and running quickly with the core tasks of Node.js.

We assume that you know a bit about what Node.js is, what it does, and why you want to use it, so this eGuide won't give you a history lesson on the background of Node.js. What this eGuide will give you, however, is a greater understanding of the key basics of Node.js so that you have a good idea of how to advance after you've read the guide. We can then point you in the right direction of what to learn next after giving you the basic knowledge to do so.

What You Need To Know About Node.js will:

- Cover the fundamentals and things you really need to know, rather than niche or specialized areas
- Assume that you come from a fairly technical background and so you understand what the technology is and what it broadly does
- Focus on what things are and how they work
- Guide you in developing an app to get you up, running, and productive quickly



# Overview

Node.js is a booming technology. It is turning out to be a popular one among the open source developers. There is a stack of articles on the web. For any naive user, to get the right one is important. That said, it is important for me to figure out a heuristic way to create a guide on Node.js.

While working on the outline of this book, I presented the Node.js introduction sessions. I found this as the best way to receive feedback about the information flow.

What inspires me about this book is the simplicity of the language and how the reader moves from the basic building blocks of JavaScript to Node.js.

This eGuide is divided into three sections.

The first section starts with the basic building blocks of JavaScript, moving toward Node.js concepts and how it works. It provides the conceptual clarity required for any beginner.

After getting an overall understanding of the Node.js architecture, we step into the creation of an application server in Node.js. This section targets beginners as well as intermediate Node.js developers. You can actually code along with the steps and get access to the best practices recommended in this section.

The last section is all about the keywords or terminology used in the preceding two sections.

You can make a note of all the highlighted keywords, which can be referred to in the last section. The structure of each keyword contains a basic description, points to remember, supportive pseudo-code, and more sources.

Let's start this learning journey and get the best possible outcomes of it.

# Table of Contents

<b>Section 1: Applying JavaScript to the Server Side</b>	<b>1</b>
<b>Learning traditional JavaScript</b>	<b>1</b>
Why does JavaScript create a single stack?	2
<b>Callback mechanism</b>	<b>3</b>
Introducing eventloop	6
Introducing Node.js	8
Eventloop revisited with Node.js	8
Single-threaded eventloop model on a server	9
<b>Why and where is Node.js used?</b>	<b>10</b>
<b>Section 2: Building a Node.js App</b>	<b>12</b>
<b>NPM community</b>	<b>12</b>
<b>Installing Node and NPM</b>	<b>13</b>
<b>Let's code</b>	<b>14</b>
Building the and configuring a server	14
Best practices	16
Picking a framework	16
Handling asynchronicity	17
Using NPM libraries	17
Debugging	17
Profiling	17
Unit testing	17
Versioning	18
Maintaining configurable settings	18
Creating API endpoints	18
<b>Future scope and the Node.js ecosystem</b>	<b>29</b>
<b>Section 3: Node.js Cheat Sheet</b>	<b>30</b>
<b>Summary</b>	<b>37</b>

# 1

## Applying JavaScript to the Server Side

The aim of this section is to introduce the basic building blocks of JavaScript and also its core working. Further, we move toward how JavaScript forms a base scripting language for Node.js. In this section, we will also provide the usefulness of Node.js and its applications.

### Learning traditional JavaScript

Early web applications were nothing but a collection of static web pages where the information was only used for viewing through browsers. The only intention was information sharing. Gradually, the need for user interactivity increased over time and so did the use of browser scripting language.

JavaScript is a popular scripting language and supports all browsers. Its usage expanded from client side to server in 2009. With the invention of Node.js by Ryan Dahl, JavaScript started running at the server side. The reasons for using it will be discussed while concluding first section.

Now let's go through the fundamentals of JavaScript required to proceed towards Node.js.

Consider the following snippet:

```
var getTotal = function(args) {  
    var result = 0;  
    for (var i = 0, len = args.length; i<len; i++) {  
        result += args[i]  
    }  
    return result;  
}
```

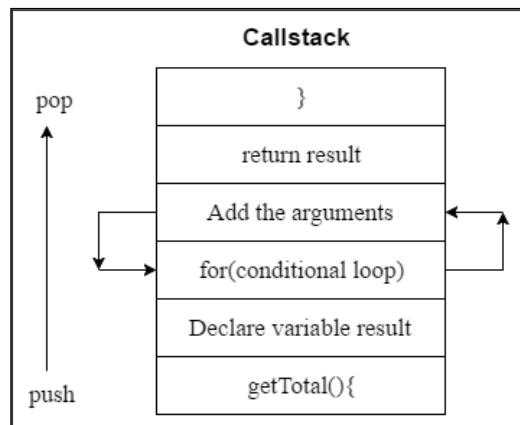
```
};  
console.log(`Output :`, getTotal([2, 2, 3]));
```

The output would be 7.



The order of assignment followed by declaration should be maintained. In the preceding code, an anonymous function is first assigned to a variable, and then it is invoked. Hence, if the function has a name, then the order is not necessary.

When the preceding code runs, the JavaScript engine creates a single **callstack** in an execution context and pushes the `getTotal` function in our case. When the `getTotal` function is called in the `console.log` method, the execution flows in following way:



Once the function returns the result, the local variables are disposed (garbage collected) and the function pops out from the stack.

## Why does JavaScript create a single stack?

JavaScript is a single-threaded language, and thus, it creates a single stack. Traditionally, JavaScript was intended for adding simple functionality and minimal runtime inside the browser. There are some issues implementing multiple threads of a JavaScript code on the same web page. The issues are outside the context of this book, so we won't be learning them here, but modern browsers have been successful in implementing web workers, a multithreaded JavaScript. Exploring the multithreaded uses of JavaScript is outside the context of this book, but I would like to recommend studying how this can be implemented in a browser.

Therefore, we can say that everything executes asynchronously in JavaScript. But what in the case of multiple functions or if one function calls the other? Let's learn it next.



JavaScript functions can be referenced to a variable within its scope, *assigned, passed as an argument, and returned* as a result. Hence, they are called **first-class functions**. In further sessions, we will study the case of callbacks.

## Callback mechanism

In this section, you will learn about callbacks and their usefulness in the asynchronous execution of code. Callback is a chief functional programming technique that provides the flexibility of passing a function as an argument to another function. Consider the following example:

```
setTimeout(function(){
  console.log(`display after sometime!`)
}, 6000)
```

The output after 6 seconds will be `display after sometime!`

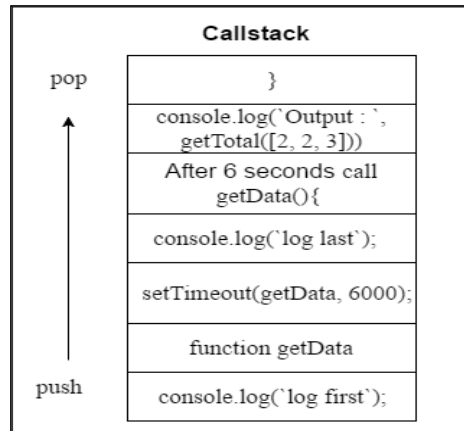


If the preceding snippet is executed in a browser console, the return value of `setTimeout` will also be visible.

The next snippet simply logs the output after 6 seconds. Let's pass the previous `getTotal` function as a parameter. This is illustrated in the following:


```
console.log(`log first`);
function getData() {
  console.log(`Output : `, getTotal([2, 2, 3])).
}
setTimeout(getData, 6000);
console.log(`log last`)
```

For the preceding code, the following callstack gets created in an execution context:

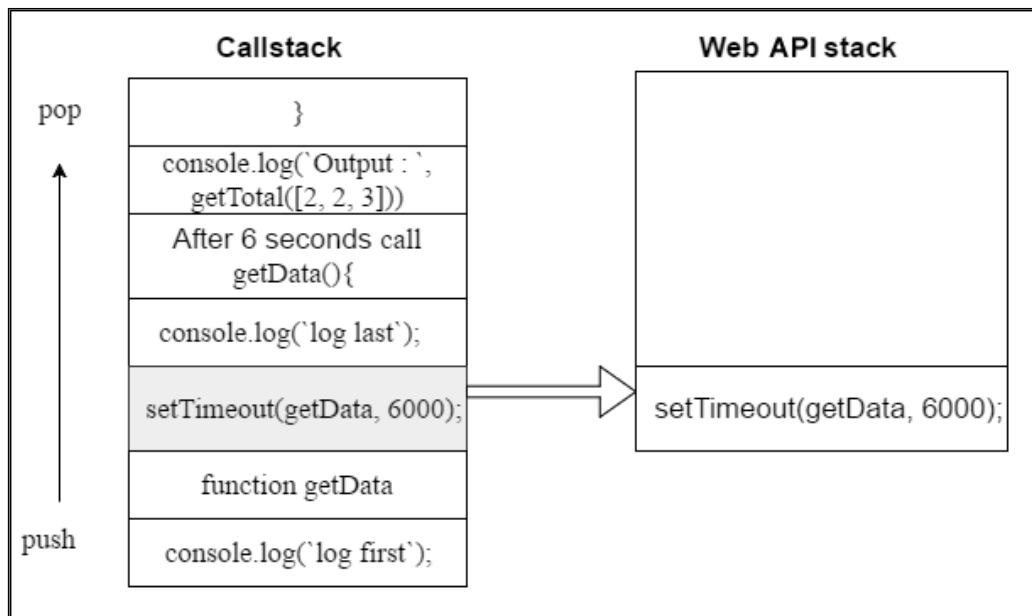


The observation in the callstack shows the output in the following sequence:

```
log first
log last
Output: 7
```

 The execution of the `getTotal` function in the preceding callstack is already explained in the previous section.

The `setTimeout` method is an inbuilt web API or document method of any browser that supports JavaScript. Therefore, the execution context of the `setTimeout` method is the browser web API and not in the developer-defined script's context. In order to visualize this scenario, consider the following diagram:

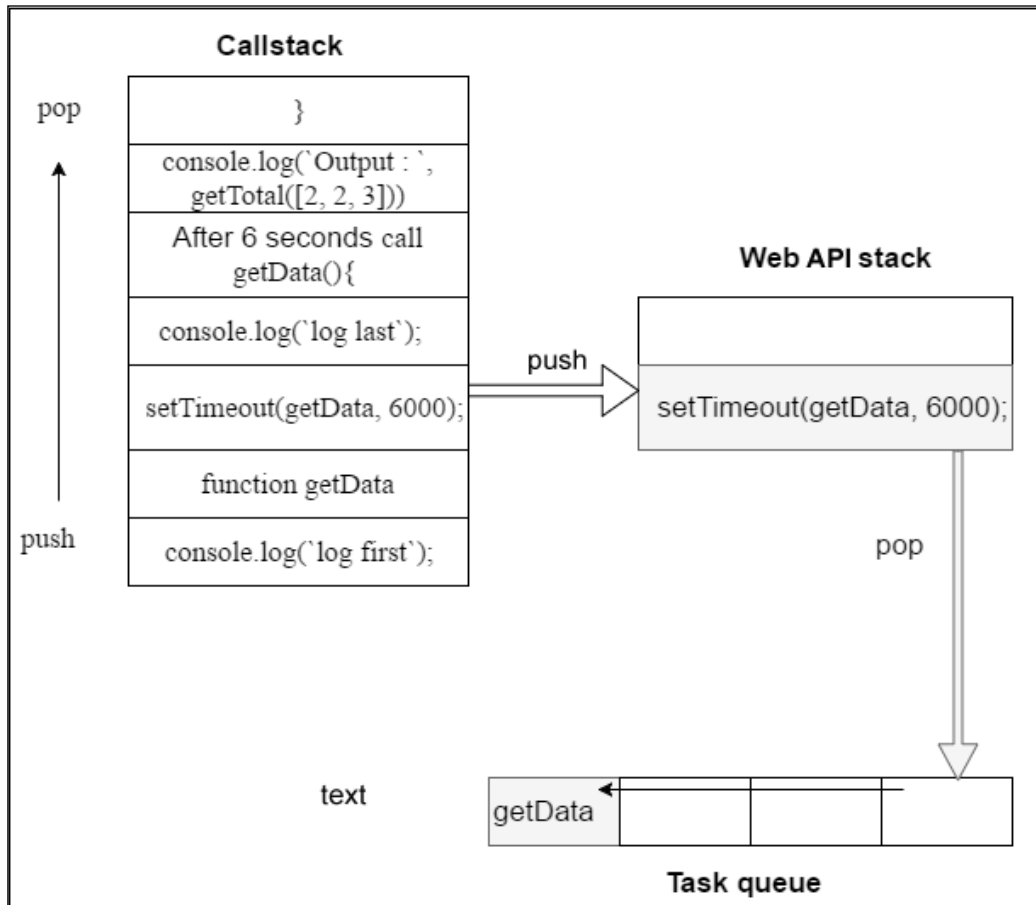


Once the operation is recognized as asynchronous (which requires web API), the method is called in a different context and the execution of the callstack continues. Hence, the `log last` is printed before the `setTimeout` method displays the result. This is why JavaScript code is said to be non-blocking.



This mechanism works similarly for XMLHttpRequest or AJAX request made from the browser. The `setTimeout` can be replaced by them to get an idea of asynchronous requests.

Once it times out in the web API stack (after 6 seconds in our case), the web API stack pushes the code to the task queue. So, considering the previous case, we have the following callstack:



What is the task queue?

The task queue contains every step that should be executed next in the callstack – the order of priority is based on a **First In First Out (FIFO)** approach.

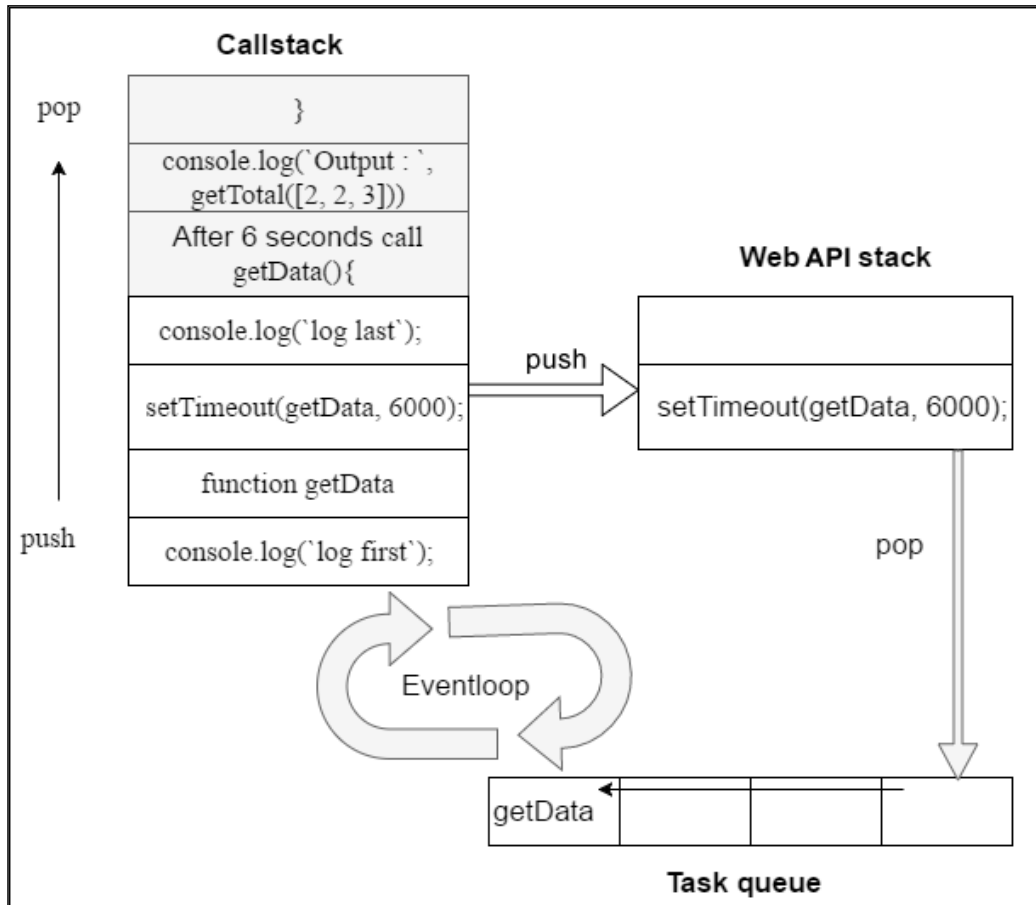
## Introducing eventloop

An eventloop is responsible for the following:

- Fetching the next task in the event queue
- Assign the task to the callstack in the execution context.



The following diagram contains a new component called **eventloop**:



On every event emitted in the browser, the eventloop keeps on checking for any tasks prioritized in the queue.

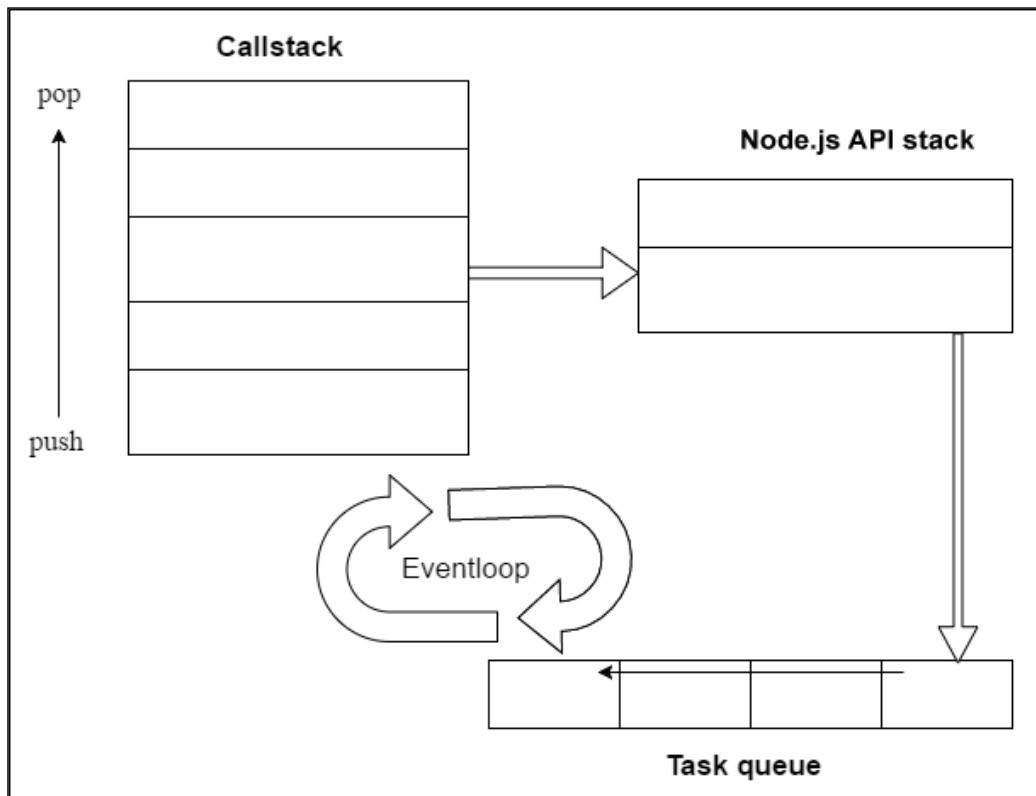
When the task is enrolled in the queue and the callstack is empty and ready for execution, the eventloop retrieves the prioritized task from the queue and performs the execution via callstack. This happens at the JavaScript runtime. Every browser engine uses a similar kind of mechanism to interpret the JavaScript code.

## Introducing Node.js

Node.js is a JavaScript runtime environment created using Chrome's v8 engine by a programmer named Ryan Dahl. According to Ryan, popular web servers such as Apache use multithreaded models for systems. Hence, the context switching between threads utilizes valuable CPU time. Moreover, the code is blocked while making any asynchronous requests or I/O operations.

## Eventloop revisited with Node.js

In the previous section, we studied the single-threaded eventloop model of JavaScript; Ryan used this concept and implemented it using server-based APIs written in C++ and JavaScript. Let's recall the eventloop diagram. Here we are reusing the base of the previous diagram for Node.js:

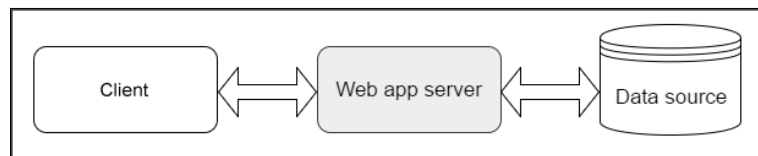


The only difference in the preceding structure is that the browser's web API is replaced with Node.js API. The aim of Ryan Dahl was to implement a complete non-blocking server system. JavaScript, being a single threaded language, was the best fit as the scripting language for Node.js. This was an evolution of JavaScript at the server side.

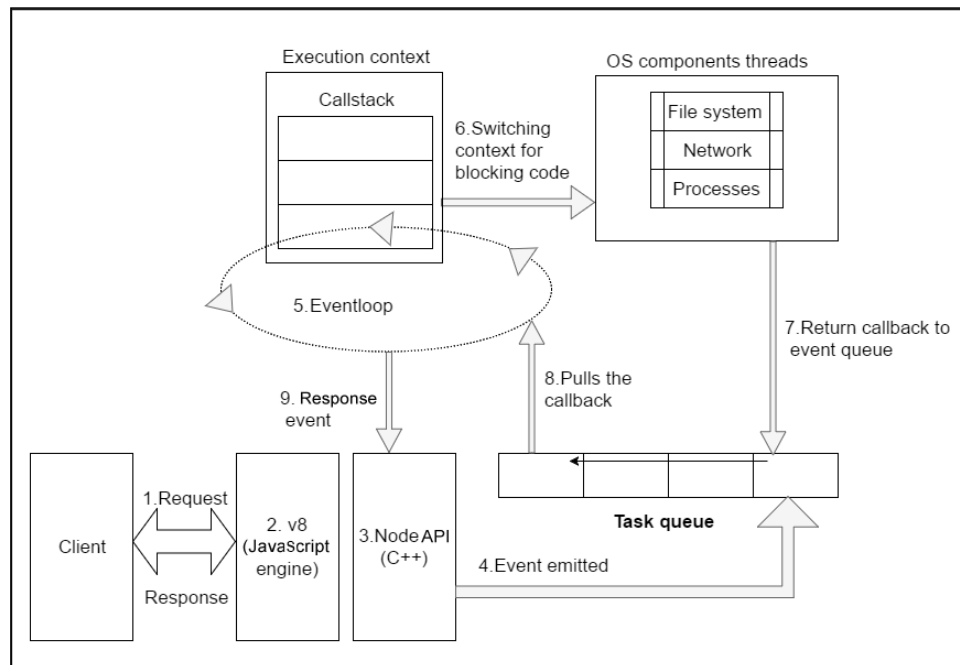
Therefore, the JavaScript browser boilerplates and libs can be reused over server side. This was the reason the term **isomorphism** came into existence with JavaScript.

## Single-threaded eventloop model on a server

Nowadays, our web applications follow a three-tier web architecture. Let's consider the following:



The web app server is where the Node.js magic happens. Let's fuse up the single-threaded event model web server and see how they operate together. The following is a component based diagram of Node.js at the server side:



The working of this architecture contains the following steps:

1. The client sends a request (consider an HTTP).
2. The Chrome's v8 engine is a **just-in-time (JIT)** compiler. Once the request is received at the server, v8 converts the JavaScript code to machine code.
3. The C++ APIs within the Node.js core provide a binding for other system level components.



Binding is basically a wrapper library so that a code written in one language can communicate with a code written in another language. This API is responsible for emitting an event.

4. Once the event is emitted, it is stored in the event queue.
5. The eventloop is responsible for fetching the event from the queue and executing it in the callstack.
6. If an event requires an asynchronous operation to be done, such as using database files, it switches its execution context to another worker thread and gets executed. This is done by **libuv**. The component libuv is detailed in the last section.
7. Once the asynchronous operation is completed, it returns the callback. The callback remains in the event queue until the callstack gets empty.
8. Once the callstack is empty, the eventloop pulls the callback from the event queue and executes it in the callstack.
9. Eventually, the event returns the data to the node API.
10. In each loop, it performs a single operation. Though the operations are performed sequentially, this single-threaded mechanized eventloop is so fast that it provides an illusion of concurrency. A single thread can utilize a single core of the system; hence, it provides better performance and minimal response time to client.

## Why and where is Node.js used?

Node.js not only performs well in building a fast and scalable network application, but also in other areas, such as:

- **Real-time and multi user apps:** Chrome's v8 engine provides fast compilation of JavaScript with an added advantage of single-threaded eventloop; it increases the response time with better performance.

- **Web API's services:** Node.js works well for web services where a flow of continuous data is required, such as games or simulators. However, it can be used with databases, but performs better when the data source used is JSON-based.
- **Delayed jobs:** Sometimes, the necessity for writing data eventually arises, for example, sending an e-mail or writing data to DB, even after the response is sent from the server. Node.js is highly recommended because it comes with modules to support this functionality.
- **Proxy:** The feature of non-blocking execution stimulates the Node.js usage as a proxy server. A node proxy server can simultaneously handle large amounts of multiple connections and is easily configurable.
- **Data streaming:** Node API provides a great stream of supportive methods. The callback mechanism of Node.js is a great way to handle the flow of stream and operate on them at the same time.
- **Minimum viable apps:** The average development time required for an application is relatively low as compared to other frameworks. The configuration required for booting a node server is minimal.
- **Monitoring and notification application:** Node.js is fast and provides real-time solutions, which makes it feasible to be used for monitoring and notification applications. Moreover, it has a rich system-level API that plays a crucial role in the surveillance of events and data.
- **Microservices:** Rather than the whole app as a single unit, dividing the modules with respect to functionality is a microservice pattern. Microservice pattern is not new. However, with the advent of the **Internet of Things (IoT)**, Node.js provides support to microservices and IoT. It enables the elements of an application to be updated and scaled separately.

The node usage list will go on increasing with time.

We had a comparative study of traditional JavaScript and Node.js. We started with a look at JavaScript in a browser and moved toward JavaScript at server. We studied the basic building blocks, such as eventloop, callstack, libuv. Finally, we studied what can be the use cases of Node.js. In the upcoming section, we will actually code and build a node application from scratch.

# 2

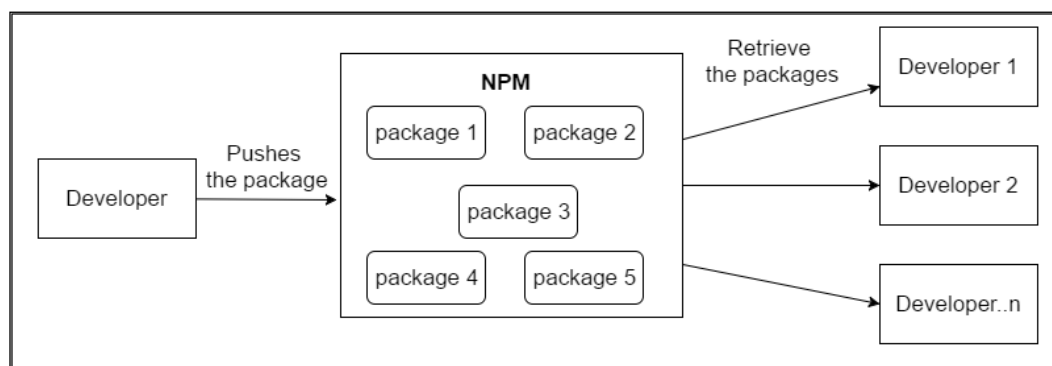
## Building a Node.js App

In this section, we will explore the community that supports Node.js and its respective modules. Further, we will walk through a guideline for the installation of Node.js and for building a Node.js API server with minimum configuration. Finally, we will end up learning some best practices and the future scope of Node.js.

This section will leave you with an overall idea of powerfully dealing with Node.js development for beginners and best practices for advanced developers. Let's start with the community that works at the roots of this technology and provides an outstanding support.

### NPM community

NPM started as a Node package manager with the advent of Node.js. Currently, it has become a package manager not only for Node.js but for all JavaScript modules. The overall intention of building the NPM was to share boilerplate (open-source code). The basic working of the NPM community is demonstrated in the following diagram:



A reusable JavaScript code is said to be a package. The developers share their own packages on the NPM repositories. NPM provides a platform to store, browse, and access these packages among developers.

Once we register our code with NPM, any developer can search for it and install it as per the requirement. These features gave rise to an NPM open source community with worldwide contributors.



NPM not only provides packages with respect to single developers but also organizations. It enables us to create an organization inside NPM and operate the package as per the license we desire.

For more details, visit <https://docs.npmjs.com/orgs/what-are-orgs>.


## Installing Node and NPM

Let's start with the installation of Node.js. As the NPM is already included in the Node.js setup, there is no need to install it separately.

Without any ado, let's start our step-by-step procedure to install Node.js and update npm:

1. Go to <https://nodejs.org>.  
Select the **DOWNLOADS** link from the menu. The link navigates to a page where a table of operating systems versus node installer formats is displayed. Download the Node.js setup according to the OS configuration and install it.
2. Open the Terminal and type `node -v` to check the installed Node.js version.  
NPM is installed as a part of the Node, but it is not necessarily up to date. This step is recommended; however, it is still optional while installation. To perform it, refer to the following code:  

```
npm install npm@latest -g
```
3. Check if npm is installed correctly by the node installer: `npm -v`.
4. Create a directory named `taskList` and go to that directory. Write the command `npm init` in the Terminal. Write the name as `task-list` and follow the other questionnaires provided by `npm init`. All the questionnaires mentioned in the Terminal are nothing but the details required to prepare the `package.json` file. Once completed, the `package.json` file is created.

 You can use `sudo npm init` in case permission is needed.

5. Now the installation of any NPM module is simple, as follows:

```
npm install <package_name><options>
```

You can view the options by hitting `npm help json` in the Terminal. However, the `options` field is normally used to save the package in `package.json`. This can be done by using `--save`.

For example, let's install a framework of Node.js, known as **Express.js**:

```
npm install express --save
```

## Let's code

Once all the packages are installed and the project structure is created, we are ready to code.

## Building the and configuring a server

Create a file called `app.js`. Follow these steps in order to write the code for creating a server:

1. The first step is to import the required module in `app.js`. In our case, we require the `http` module of the node API. We use the `require` function to import a module. We also store an instance of the module in a variable so that it can be used further. Consider the following line of code:  

```
var http = require("http");
```
2. The next step is to use the `createServer` method of the `http` instance. A port is a mandatory requirement to create a server. We can specify that in the `listen` method. It is demonstrated in the following:  

```
http.createServer(requestListener).listen(8081);
```
3. The `createServer` method requires a callback as a parameter. In the following code, we have used `requestListener` as a callback function. This callback is used to handle the requests made by the client and provide a response back to the client. Whenever a server is hit by an `http` request, this callback is called. This can be specified as follows:



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

var port = 8081;
http.createServer(requestListener).listen(port);
console.log("Server is listening on", port)

function requestListener(req, res){
  res.writeHead(200, {'Content-Type': 'text/plain'});
  console.log("Request recieved, responding now ...");
  res.end("Hello");
}
```



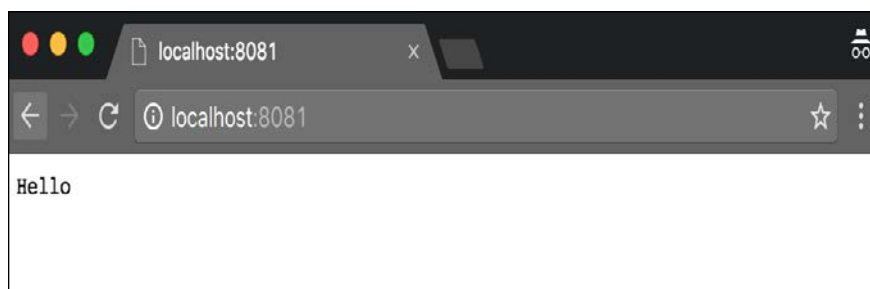
The `requestListener` function has two parameters: a *request instance* as `req` and *response instance* as `res`.

The `writeHead` method provides API abstraction to write the content type of response to be sent with the status code. In the preceding snippet, the success code 200 is used. Finally, an `end` method of the response instance is used to send the data to the client.

4. That's it. Our node server is ready for launch. Go back to the Terminal and execute the following command: `node app.js`, as seen in the following screenshot:

```
admins-MacBook-Pro-2:todo1ist brunodmello$ node app.js
Server is listening on 8081
```

5. Open a browser and hit the URL, `http://localhost:8081/`. The following output will be displayed:



## Best practices

Every game has its own rules. Although not mandatory, best practices are some standards that are recommended and followed with respect to the development of an application. Learning about them will be of great help. The following subsections cover some recommended best practices.

## Picking a framework

A framework can be said to be a predefined structure of code with some abstraction, and using them makes the development quite fast and easily maintainable. Such standard frameworks are already available within the node community. The only fuss is a matter of choice.

To make the choice easier, frameworks can be categorized as follows:

- Configuration-based frameworks
- Syntactic/semantic style-based frameworks
- Unopinionated frameworks

All these types are explained in detail in the last sections. An example of a framework can be Express.js, which we installed earlier. Let's apply the abstraction in our `app.js`. The following comparison of codes provides some clarity on using a framework over the core Node.js API:

<pre>var http = require("http");  var port = 8081; http.createServer(requestListener). listen(port); console.log("Server is listening on", port)  function requestListener(req, res){   res.writeHead(200, {'Content-Type': 'text/plain'});   console.log("responding now");   res.end("Hello"); }</pre>	<pre>var express = require('express');  var app = express(); var port = 8081; app.use(requestListener).listen(port); console.log("Server is listening on", port)  function requestListener(req, res){   console.log(" responding now");   res.send('Hello'); }</pre>
--	--

As shown in the preceding comparison, a framework provides abstraction of the core API. The functionality is handled more efficiently.

## Handling asynchronicity

By default, either the callbacks or the event emitters are used to handle asynchronous behavior in Node.js. As the complexity of the asynchronous functions in the code increases, the usage of callbacks also increases. As a result, it simply creates a code structure that is not easy to understand. Using too many callbacks gives rise to *callback hell* and to avoid this, the *promises library* is used. Note that the callback hell is not an issue or a bug. We will study callback hell and promises library in the last section. Nowadays, RxJS is a new concept to handle asynchronous behavior.

## Using NPM libraries

Libraries and utility modules on NPM are already maintained and optimized by developers. So, rather than writing boilerplate code of our own, using the library is recommended with respect to its use and performance efficiency. We will be using the utility library called *Lodash.js* in our next sub section.

## Debugging

The callback mechanism can be overwhelmed with errors. Hence, finding the real cause can be tricky sometimes. The node community provides some tools such as *node-inspector*, which helps to manage all the debugging.

## Profiling

In an app where performance throughput is the key, profiling is a necessity. Profiling provides us with statistic information about the current memory utilization with respect to the load or the simple execution of a function. In a JavaScript code, a memory leak can be detected by profiling tools such as *memwatch*. The details about *memwatch* are provided in the last section.

## Unit testing

It is said that writing the unit test of a module comes prior to writing the actual functionality of the module. Writing unit test cases makes sure you provide a collection of tests with respect to each part of the module. Two prominent unit test techniques are as follows:

- **Test-driven development (TDD):** Test-driven development as a process consists of writing a unit test, executing it, and verifying whether the test passes or fails. The a JavaScript community provides decent TDD suites.

- **Behavior-driven development (BDD):** Behavior-driven development testing bends towards testing the responses of a functionality. BDD testing is done by an assertion library. This library provides a language to test the expected behavior of code.

So, a combination of both TDD and BDD techniques is applied to test the expected results. We will be implementing both the techniques in the next subsection for creating an API in Node.js.

## Versioning

Versioning is a great technique to maintain an application over time. All the applications in JavaScript registered with the NPM follow version. The version control paradigm stimulates to maintain a track list of all the features, dependencies, and fixes in a file. For instance, the `change.log` file in the GitHub repository is a great example. It is really worth managing the project by versioning it.

Semantic versioning specifications have become the core part of Node.js development. It is also known as **semver**. The NPM provides packages that stimulate semver practices.

For more details, check out <http://semver.org/>.

## Maintaining configurable settings

JSON is extensively used in Node.js. The JSON files in a Node.js project are normally used for configuration purposes. The purpose of storing separate configuration storage is widely used for service oriented architecture. No sooner do we require a module than it's ready for use. One of the popular NPM packages is `node-config`. It is reliable and easy to use. Please visit the following URL for its installation and usage:

<https://github.com/lorenwest/node-config>

In the previous subsections, we studied the basic building blocks of Node.js server and the implementation of best practices; now let's start creating an application API.

## Creating API endpoints

In the following steps, we will create an API for a simple to-do list, considering some recommended practices. We are going to use `nodemon`, which is an NPM module. It is responsible for continuously supervising the node development. The source for `nodemon.js` is provided in the last section.

Globally, install nodemon as `npm i nodemon -g`.

Run it as `nodemon app.js` and follow these steps:

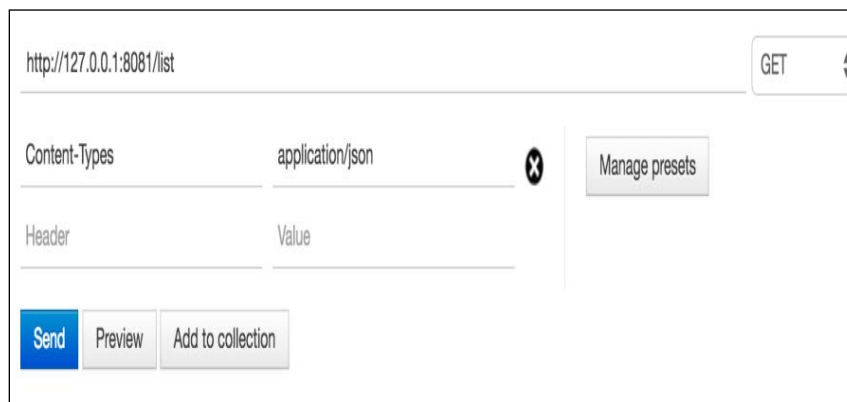
1. Create a server instance in Express and run it (we have already installed Express during the installation process). Express.js is a simple and minimalist framework for devising an entire web application in Node.js. For more details and the source of Express.js, please refer last section of this guide, *Node.js Cheat Sheet*. So, let's start our code in Express:

```
var express = require('express');
var port    = 8081;
var app     = express();
app.listen(port);
```

2. Create a route handler for getting the list data. The data storage currently used is an in-memory array called `taskList` (for demo purposes). We can use persistent storage, such as database, files, and so on. In the following code, a callback handler is assigned to a route `/list`. The route name and callback handler are parameters in the `get` method of the Express app instance. Add the following code to `app.js`:

```
var taskList = [];
app.get('/list', function(req, res){
  res.send(taskList);
})
```

3. To check the API code, we can use Postman. **Postman** is a Chrome extension to send a request to a server. Once all the required information is filled as shown in the following screenshot, hit the **Send** button to request. We will receive an empty array.



4. The next step is to add data to the array list. This can be done using the `post` method, as shown in the following steps:
  1. Use the `post` method of the Express app.
  2. Extract the body from the request using the `body-parser` middleware.

We can use `npm install body-parser --save`. The following screenshot shows the successful completion of the body parse installation:



```
admins-MacBook-Pro-2:todolist brunodmello$ npm i body-parser --save
npm WARN package.json todolist@1.0.0 No description
npm WARN package.json todolist@1.0.0 No repository field.
npm WARN package.json todolist@1.0.0 No README data
body-parser@1.15.2 node_modules/body-parser
├─ bytes@2.4.0
├─ content-type@1.0.2
├─ depd@1.1.0
├─ qs@6.2.0
├─ iconv-lite@0.4.13
├─ on-finished@2.3.0 (ee-first@1.1.1)
├─ http-errors@1.5.0 (setprototypeof@1.0.1, inherits@2.0.1, statuses@1.3.0)
├─ raw-body@2.1.7 (unpipe@1.0.0)
├─ debug@2.2.0 (ms@0.7.1)
└─ type-is@1.6.13 (media-typer@0.3.0, mime-types@2.1.11)
admins-MacBook-Pro-2:todolist brunodmello$
```

Append the following code to `app.js`:

```
var bodyParser = require('body-parser');
app.use(bodyParser.json());
```

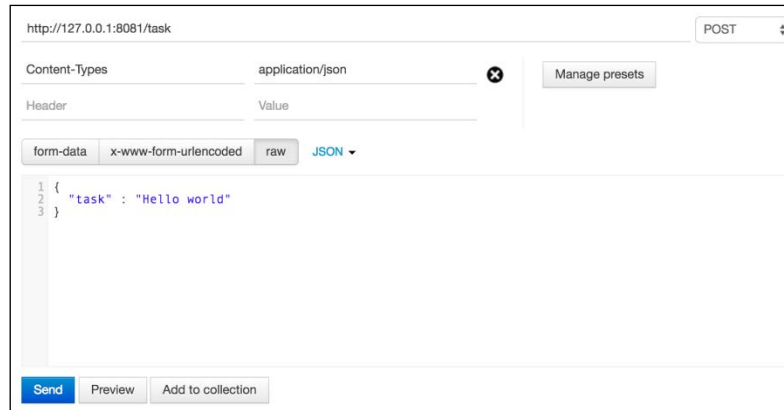
Further, let's continue the preceding steps:

3. Add data to array.
4. Finally, send the response as an array list.

Here is the code for pushing the data onto the list:

```
app.post('/task', function(req, res){
  taskList.push(req.body.task);
  res.send(taskList);
})
```

Once completed with the coding steps, let's see how it works by requesting our server using Postman:



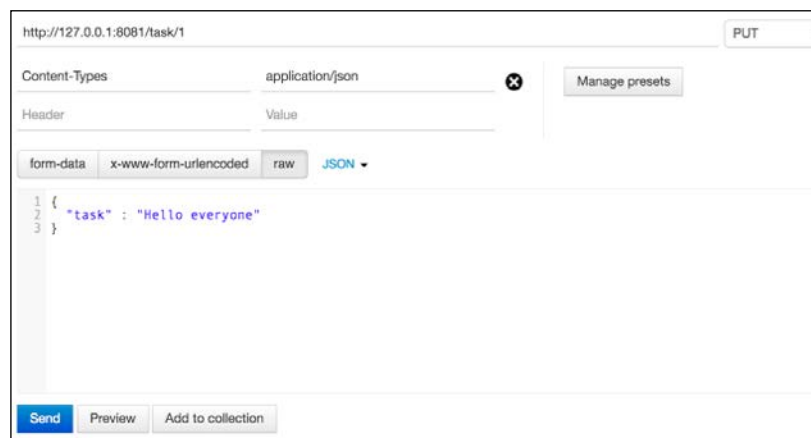
Within no time, we receive the list of tasks as an output.

5. Now, we can create an API to update and delete the data. Here are the following structures for PUT and DELETE API endpoint implementations:

The snippet of `app.put` API block is as follows:

```
app.put('/task/:task_index', function(req, res){
  var taskIndex = req.params.task_index;
  taskList[taskIndex] = req.body.task
  res.send(taskList);
});
```

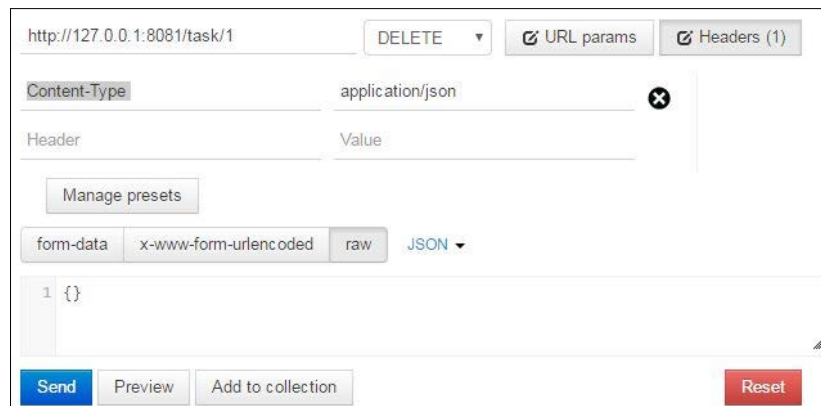
Its corresponding Postman screenshot is as follows:



The snippet for the delete endpoint is as follows:

```
app.delete('/task/:task_index', function(req, res){
  var taskIndex = req.params.task_index;
  taskList.splice(taskIndex, 1)
  res.send(taskList);
})
```

The corresponding Postman screenshot is as follows:



- Here we go; our basic API endpoints are ready. Let's enhance our prototype with some basic utility libraries for validation purposes. We will be installing the `lodash` library. The `lodash` is a utility library that lets us efficiently handle operations on data stores such as variables, objects and arrays. It is installed using the following command: `npm install lodash --save`

Once installed, we can require the library as follows:

```
var _ = require('lodash');
```

Consider the `post` method API we created before. If the task received in the body message is empty, a null value will be added to `taskList`. So, to validate this we can use the `isEmpty` method of `lodash`:

```
app.post('/task', function(req, res){
  var task = req.body.task;
  if(_.isEmpty(task)){
    return res.status(422).send("Task is empty");
  }
  taskList.push(req.body.task);
  res.send(taskList);
})
```





To learn more about `lodash`, please visit its site at <https://lodash.com/docs/4.15.0>.

- Now if we review the `PUT` API, it needs the same kind of validation. So, rather than putting it differently for each API, we can write the validation functionality in a middleware and reuse it wherever desired. Writing a middleware using `Express.js` is really easy. Let's consider the following snippet:

```
function validation Middleware(req, res, next){
  if(_.isEmpty(req.body.task)){
    return res.status(422).send("Task is empty");
  }
  return next();
}
```

We can use the following code to link the middleware to the callback handler. Both the endpoints, that is, `add task` and `edit task` as provided in the following contain a middleware. It is responsible for checking whether the task is present in the body of the client request:

```
app.post('/task', [validationMiddleware, function(req, res){
  var task = req.body.task;
  taskList.push(task);
  res.send(taskList);
}])

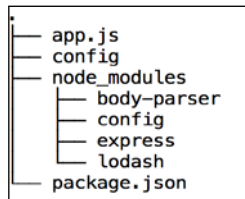
app.put('/task/:task_index', [validationMiddleware, function(req,
res){
  var taskIndex = req.params.task_index;
  taskList[taskIndex] = req.body.task
  res.send(taskList);
}])
```



The third parameter of a middleware is a function defined as `next`. When we call the `next` function, the execution moves to the next middleware callback. The eventloop model studied in the first section fetches the next callback from the event queue to the execution stack. Understanding the `next` function is recommended so that we can decide when to continue or skip an eventloop.

8. Let's apply one of the best practices of maintaining the configuration details. Including this step, we give the developer an essence of maintaining the configuration files for different environments periodically. To work on it, let's use NPM's `config` module as follows:
  1. The installation command for `config` is as follows:  

```
npm install config --save
```
  2. Follow the following map to get an idea of the directory structure. Make a `config` directory using the `mkdirconfig` command.



3. Go to the `config` directory and create a `local.json` file. Write the following JSON into it:

```
{
  "host" : "localhost",
  "port" : 8081
}
```
4. We need to link the `config` module to our application—to do this, go ahead and modify the `app.listen` method as shown:

```
var config = require('config');
//Modified listening method
app.listen(config.port);
```

Hence, we created a `config` file to store all of the configuration credentials and server details. We can create a file with respect to the environment we use; for instance, it can be development, staging, or production.

For more details, visit <https://github.com/lorenwest/node-config>.

9. Now is the time to write some unit tests for our app. The Mocha test suite and Chai assertion library are popularly used. For TDD we can use the Mocha testing framework and for BDD testing, we will be writing assertions using Chai. Although Mocha provides the assertions functionality of BDD, the Chai library specializes it and provides a wider set of API. It has great online forums and discussions too.

Here are the steps for installing and configuring our unit test suites:

1. Installation of the Mocha and Chai is simple as any other NPM packages, which is as follows:

```
npm install mocha --save-dev
```

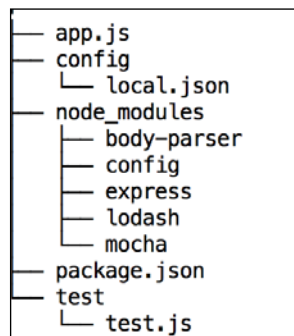
2. Once done, install Chai.js:

```
npm install chai --save-dev
```



The `--save-dev` option registers the package under development dependencies in `package.json`. All the packages under development dependencies are not installed while moving in the production environment. There is no use of development dependencies in production. Hence, the list of development NPM packages is maintained separately.

3. Mocha requires the `test` directory and any `.js` file inside it as a prerequisite, or else it gives an error. So, create a directory `test` containing a file named `test.js` so as to write all the unit test scripts in it, and we are left with the following directory structure:



10. Unit testing implementation is as follows:

The unit test coding starts with the `describe` block. It is written as follows:

```
describe('---Testing the task list api---', function(){
  it('POST: Task in list', function(done) {
    done();
  });
})
```

The `describe` block lets us test separate modules with respect to an application. It describes what we need to test.

An `it` block contains the actual test scenarios that need to be coded for specific functionality. We can write multiple nested `describe` and `it` blocks.



The `done` parameter of the callback in the `it` function controls the end of a test scenario in its block. It is invoked once all the test cases are written.

Up next, we need to write assertions for our `POST` API response. To do so, we need an instance of the required Chai library and use the interfaces provided by Chai.js. The following snippet provides it:

```
var chai = require('chai');
var expect = chai.expect;
describe('---Testing the task list api---', function() {
  it('POST: Task in list', function(done) {});
})
```

Our test suites don't provide an API for requesting the app server. We also need a library for this purpose. Here, we are installing the `npm request` modules development dependency. Once done, let's implement the following code:

```
var chai = require('chai'),
    expect = chai.expect,
    should = chai.should();
var request = require('request');
var config = require('config');
//Using config module to create a local domain
GLOBAL.domain = 'http://' + config.host + ':' + config.port;
describe('---Testing the task list api---', function() {
  it('GET: Task list', function(done) {
    /**
     * Follow the request module documentation.
     * go to : https://github.com/request/request
     */
    var options = {
      url: domain + '/task',
      headers: {
        "Content-Type": "application/json"
      },
      json: {
        task: "Hello world"
      }
    };
    request(options, function(err, res, body) {
      done();
    });
  });
})
```

```
    }  
  };  
  request.post(options, function(error, response, body) {  
    console.log("we got response", body)  
    done();  
  });  
});  
})
```

Run the `mocha` command to get the following output:

```
admins-MacBook-Pro-2:tasklist brunodmello$ mocha  
  
---Testing the task list api---  
we got response [ 'Hello world' ]  
✓ GET: Task list (114ms)  
  
1 passing (124ms)
```

Next, let's write some assertion test cases in our response callback of the preceding code. Consider the following snippet to illustrate it:

```
request.post(options, function(error, response, body) {  
  //Should be conditions  
  response.statusCode.should.equal(200);  
  //Expected conditions  
  expect(body).to.be.a('array');  
  expect(body).to.include(options.json.task);  
  done();  
});
```

Now run mocha to receive and check if the test cases are passed, as shown in the preceding screenshot. In case the response received is not as expected, or if the response status code changes, the test case fails and prints 0 passing.

Let's try this out by sending an empty task data in options, as follows:

```
var options = {
  url: domain + '/task',
  headers: {
    "Content-Type": "application/json"
  },
  json: {
    task: ""
  }
}
```

Run mocha, and the output is as follows:

```
admins-MacBook-Pro-2:tasklist brunodmello$ mocha

---Testing the task list api---
1) GET: Task list

0 passing (60ms)
1 failing

1) ---Testing the task list api--- GET: Task list:

  Uncaught AssertionError: expected 422 to equal 200
    + expected - actual

    -422
    +200

    at Request._callback (test/test.js:32:31)
    at Request.self.callback (node_modules/request/request.js:187:22)
    at Request.<anonymous> (node_modules/request/request.js:1044:10)
    at IncomingMessage.<anonymous> (node_modules/request/request.js:965:12)
    at endReadableNT (_stream_readable.js:905:12)
```

This gives an overall idea of the expected response and how it fails in case we pass an invalid payload in the request. Similarly, we can write unit tests for other API endpoints.



We can use task runners such as grunt, broccoli, gulp, and so on, to automate the test scenarios in a project. Here the test scenarios can be referred to as multiple describe blocks in multiple files, segregated with respect to functionality.

Congratulations! With these 10 steps we completed all the basic building blocks required for the development of Node.js projects.

## Future scope and the Node.js ecosystem

The usage of JavaScript on the frontend as well as the backend made it known as isomorphic. This provides an advantage to developers to learn only a single language and code on both the aspects of the development stack. Due to its pros such as being single-threaded and its performance efficiency, the overall usage has increased a lot. The open source communities will support Node.js for it has proven to be an evolution of JavaScript at the server side.

Using JavaScript everywhere is the next big thing. The open source JavaScript communities are growing at a great speed. With such new technology and an amazing community support, the possibilities of new enhancements will open up in Node.js. Atwood's law states, "what can be written in JavaScript will eventually be written in JavaScript." Node.js finds applications in various areas, such as mobile operating systems, Internet of Things (IoT), building continuous activity flows, backend as a service, and so on. Such sophisticated technology extends and grows continuously. Some bigger giants such as NASA use Node.js for spacesuit processing and logistics.

References: <https://twitter.com/CollinEstes/status/741994459349417984>.

Case study: [https://nodejs.org/static/documents/casestudies/Node\\_CaseStudy\\_Nasa\\_FNL.pdf](https://nodejs.org/static/documents/casestudies/Node_CaseStudy_Nasa_FNL.pdf).

To summarize this section, we walked through the step-by-step procedure of installing Node.js and building a server into it. We discussed the best practices and created a Node.js API project of `taskList`. After a follow-up, you could confidently kick start the Node.js development process. You will get a brief idea regarding the terminologies of and architecting a Node.js app.

# 3

## Node.js Cheat Sheet

In this section, we will discuss the fundamental set of references required for Node.js. Most of the following terminologies are references used in this guide:

- **[libuv]**
  - @type: Built-in C library.
  - @description: The libuv library is responsible for handling the asynchronous behavior of an event life cycle in the system. It is written in C. It maintains a pool of threads to handle asynchronous requests such as IO and network-related operations.
  - @References

Design overview:  
<http://docs.libuv.org/en/v1.x/design.html>  
Code overview:  
<https://github.com/libuv/libuv>.
- **[package.json]**
  - @type: JSON file.
  - @description: The package.json maintains a registry of modules installed with NPM. The name and version fields are important in the overall document. They uniquely identify an application.
  - @related-keypoint: The packages installed without save options are not stored in package.json for a project. For example: `npm install express --save`.
  - @References

Detailed overview:  
<https://docs.npmjs.com/files/package.json>.



- **[require]**

- @type: Global variable.
- @description: The `require` is just a wrapper around `Module._load`. `Module._load` is responsible for loading modules that are exported.

When a module is loaded for the first time, it is cached. The next time, `Module._load` checks whether the module is present in the cache. If not, it again loads a new copy. Caching helps in reducing the file reads, and this speeds up the application significantly.

- @pseudo-code:

```
//file :example.js
module.exports = function(){
  console.log("Hello everyone!")
}
```

```
//Output: "Hello everyone"
require('./example')();
```

- @related-keypoint: The callback function is a subscription of the leak event. Whenever the leak event is fired, the callback gets a call.
- @References

Detailed references:

<http://fredkschott.com/post/2014/06/require-and-the-module-system/>

Overview reference:

<https://nodejs.org/api/modules.html>.

- **[http module]**

- @type: JavaScript module.
- @description: The `http` module provides all the methods required to handle an HTTP server. We can use the `http` module by requiring as follows:
- @pseudo-code:

```
var http = require('http');
http.createServer([requestListener]) // returns a server
instance.
//Server instance can be further use to listen on specified
port
http.createServer([requestListener]).listen(3000)
```

- @related-keypoint: The http module is designed to handle any feature of the HTTP protocol on server. requestListener is called whenever the request hits the server.

- @References:

API overview:

[https://nodejs.org/dist/latest-v6.x/docs/api/http.html#http\\_http](https://nodejs.org/dist/latest-v6.x/docs/api/http.html#http_http)

- **[expressJS][unopinionated framework]**

- @type: JavaScript module.
- @description: Express.js is the most popular web framework used till date. It is said to be an unopinionated framework for Node.js because it has flexibility and support as the backbone for any kind of web applications.

- @pseudo-code:

```
var express = require('express');
var app     = express();
app.listen(8081);
```

- @References:

Guide overview:

<https://expressjs.com/>

- **[request]**

- @type: Request instance.
- @description: It occurs to us as the parameter of a callback provided in the creation of a server or its route in Node.js. It is emitted every time a request arrives at the server. With respect to Node.js 6, it is an instance of the http.Server class.

- @pseudo-code:

```
function requestListener(req, res){
  console.log(" Request recieved now", req);
  res.send('Hello');
}
```

- @related-keypoint: Though the Request instance is a mutable stream, it is highly recommended to create a new key inside the instance rather than using an existing instance.

- @References:

Guide overview:

[https://nodejs.org/api/http.html#http\\_class\\_http\\_clientrequest](https://nodejs.org/api/http.html#http_class_http_clientrequest).

- **[response]**

- @type: Response instance.
- @description: It occurs to us as the parameter of a callback provided in the creation of a server or its route in Node.js. Response represents an instance to be sent back to the client. It is passed with respect to Node.js 6. It is an instance of the `http.ServerResponse` class.
- @pseudo-code:
 

```
functionrequestListener(req, res){
  console.log(" responding now", res);
  res.send('Hello');
}
```
- @related-keypoint: Though the Response instance is a mutable stream, it is highly recommended to create a new key inside the instance rather than using an existing instance.
- @References:

Guide overview:

[https://nodejs.org/api/http.html#http\\_class\\_http\\_clientrequest](https://nodejs.org/api/http.html#http_class_http_clientrequest).

- **[configuration based frameworks]**

- @type: Framework.
- @description: As its name indicates, the application API endpoints created in this type of framework are configured and then assigned.
- @related-keypoint: Such types of frameworks are highly recommended for a big-structured application or SOAs.
- @References:

Example overview:

<http://sailsjs.org/> and <http://hapijs.com/>.

- **[syntactic/semantic styled based framework]**

- @type: Framework.

- @description: This type of framework adheres to a systematic way of code structuring and maintainability. It boils down the code to some abstract or chainable methods.

- @References

Example overview:

<http://koa.js.com/>.

- **[callback hell]**

- @type: Programming paradigm.
- @description: A calling thread will not halt while making any non-blocking requests. This is possible due to callbacks. Callback hell is a scenario when multiple non-blocking requests are made. As the complexity increases, the code structure moves gradually forward such that it is difficult to understand the logic behind it.

- @pseudo-code:

```
readFileContents("sample.txt", function (text){
  console.log("First log");
  readFileContents("sample2.txt", function (text2){
    console.log("Second log");
    writeFileContents("sample3.txt", text+text2, function(){
      console.log("Last log");
    });
  });
});
```

- @related-keypoint: Callback hell can be tackled using promises, generators and asynchronous functions.

- @References:

Overview references:

<http://callbackhell.com/>.

- **[Promises]**

- @type: Library.
- @description: Anything returned or thrown (throw error) in the callback is not handled. Moreover, callback hell comes into picture if the complexity of callbacks increases. To avoid all these drawbacks, the promise library is used as a solution.

- @pseudo-code:
 

```
readFileContents("sample.txt", function (text){
  console.log("First log");
  readFileContents("sample2.txt", function (text2){
    console.log("Second log");
    writeFileContents("sample3.txt", text+text2, function(){
      console.log("Last log");
    });
  });
});
```
- @References:
 

Overview references:

<http://callbackhell.com/>.
- **[RxJS]**
  - @type: Library.
  - @description: In a scenario of asynchronous streaming data, that is, when the data object is not in the form of a single chunk (for example: promises, where we either receive success or catch an error), RxJS is used. The data is considered as streams and RxJS provides an *observable* to listen to the streams.
  - @pseudo-code:
 

```
Observable.create(<subscriberFunction>)
```
  - @related-keypoint: The subscriber function is responsible for continuously observing the response and handling it.
  - @References:
 

Overview references:

<https://github.com/Reactive-Extensions/RxJS>.
- **[node-inspector]**
  - @type: Library.
  - @description: This tool provides an environment for debugging the application code written in Node.js. It uses websockets. It is fast and reliable. It enables setting breakpoints, the inspection and display of variables objects, and so on, step-over and step-in code, and pause and resume code executions. In short, all the debugging features are provided by an IDE.

- @related-keypoint: The Debugger keyword, used in browsers to debug the JavaScript code, can also be used to set the breakpoints in code. The node inspector can be installed like any other NPM package.
- @References:  
  
Overview references :  
<https://github.com/node-inspector/node-inspector>.
- **[nodemon]**
  - @type: Library.
  - @description: This is yet another tool for monitoring the changes and reloading the Node.js app instance, while running.
  - @References:  
  
Overview references:  
<https://github.com/remy/nodemon>.
- **[memwatch]**
  - @type: Library.
  - @description: Profiling is a highly recommended best practice to perform during Node.js development. A majority of the server crashes in Node.js are due to memory leaks. So, the optimization of code with an appropriate tool is necessary. memwatch detects the leaks and provides information related to them.
  - @pseudo-code:  

```
memwatch.on('leak', function(info) { ... });
```
  - @related-keypoint: The callback function is a subscription to the leak event. Whenever the leak event is fired, the callback gets a call.
  - @References:  
  
<https://github.com/marcominetti/node-memwatch>.
- **[Postman]**
  - @type: Chrome extension.
  - @description: Node.js API development can become a tedious job without using an API requesting tool. Postman provides all the functionality to request an API server.

- @References:

Overview reference:

<http://www.github.com/a85/POSTMan-Chrome-Extension/wiki>.

## Summary

The sections in this guide provide a gradual learning experience from core JavaScript to JavaScript at the server side. In the first section, we started with a simple function in JavaScript and how a callstack is created, further moving towards the single-threaded eventloop model. The application of such a model at the server side makes Node.js incredible.

In the second section, we created a to-do application using a step-by-step procedure, followed by best practices and libraries provided by the Node.js ecosystem that we need to know while bootstrapping an app in Node.js.

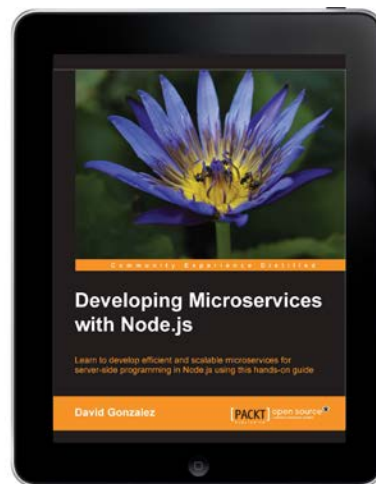
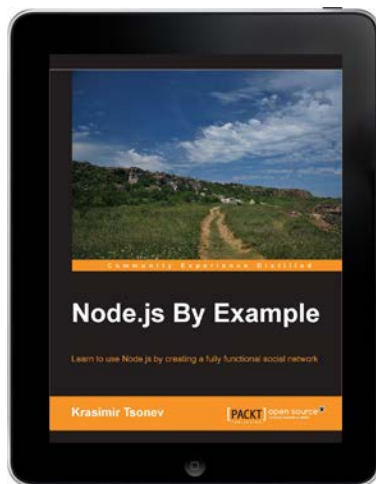
The last section completes the guide by referring to different terminologies and keywords used in the Node.js community.

Congratulations on making it this far. Well, this is just the start; as it is rightly said, "*Mastery is nothing but continuous learning process*". We would recommend you to follow up the Node.js developer communities and keep the good work on.

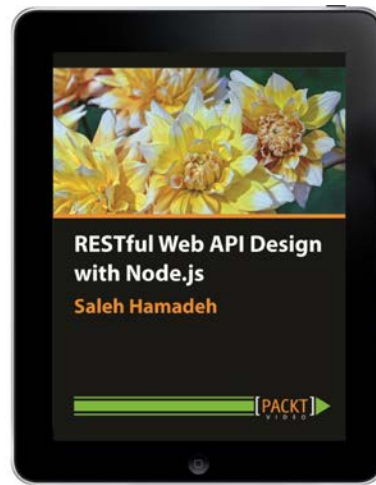
# What to do next?

## Broaden your horizons with Packt

If you're interested in Node.js, then you've come to the right place. We've got a diverse range of products that should appeal to budding as well as proficient specialists in the field of Node.js.







To learn more about Node.js and find out what you want to learn next, visit the Node.js technology page at <https://www.packtpub.com/tech/Nodejs>.

If you have any feedback on this eBook, or are struggling with something we haven't covered, let us know at [customercare@packtpub.com](mailto:customercare@packtpub.com).

Get a 50% discount on your next eBook or video from [www.packtpub.com](http://www.packtpub.com) using the code:

