

WA2452 Node.js Software Development



Web Age Solutions Inc.
USA: 1-877-517-6540
Canada: 1-866-206-4644
Web: <http://www.webagesolutions.com>

The following terms are trademarks of other companies:

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

IBM, WebSphere, DB2 and Tivoli are trademarks of the International Business Machines Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

For customizations of this book or other sales inquiries, please contact us at:

USA: 1-877-517-6540, email: getinfousa@webagesolutions.com

Canada: 1-866-206-4644 toll free, email: getinfo@webagesolutions.com

Copyright © 2017 Web Age Solutions Inc.

This publication is protected by the copyright laws of Canada, United States and any other country where this book is sold. Unauthorized use of this material, including but not limited to, reproduction of the whole or part of the content, re-sale or transmission through fax, photocopy or e-mail is prohibited. To obtain authorization for any such activities, please write to:

Web Age Solutions Inc.
439 University Ave
Suite 820
Toronto
Ontario, M5G 1Y8

Table of Contents

Chapter 1 - Introduction to Node.js	9
1.1 What Is Node.js?.....	9
1.2 Application of Node.js.....	9
1.3 Installing Node.js and NPM.....	10
1.4 "Hello, Node World!".....	11
1.5 How It Works.....	11
1.6 Built on JavaScript: Benefits.....	12
1.7 Traditional Server-Side I/O Model.....	12
1.8 Disadvantages of the Traditional Approach.....	12
1.9 Event-Driven, Non-Blocking I/O.....	13
1.10 Concurrency.....	13
1.11 Using Node Package Manager (NPM).....	14
1.12 Express.....	14
1.13 Summary.....	15
Chapter 2 - Module and Dependency Management.....	17
2.1 Nature of a Node.js Project.....	17
2.2 Introduction to Modules.....	17
2.3 A Simple Module.....	18
2.4 Using the Module.....	19
2.5 Directory Based Modules.....	19
2.6 Example Directory Based Module.....	20
2.7 Using the Module.....	20
2.8 Making a Module Executable.....	21
2.9 Core Modules.....	22
2.10 Loading Module from node_modules Folders.....	22
2.11 Dependency Management Using NPM.....	23
2.12 Installing a Package.....	23
2.13 About Global Installation.....	24
2.14 Setting Up Dependency.....	25
2.15 Package Version Numbering Syntax.....	25
2.16 Updating Packages.....	26
2.17 Uninstalling Packages.....	27
2.18 Alternate Dependency Management.....	27
2.19 Summary.....	28
Chapter 3 - The File System Module.....	29
3.1 Introduction.....	29
3.2 Basic File Manipulation.....	30
3.3 Getting File/Directory Meta Data.....	30
3.4 Read an Entire File.....	31
3.5 The Buffer Class.....	31
3.6 Writing to a File.....	32
3.7 Reading in Chunks.....	32
3.8 Writing in Chunks.....	33
3.9 The open() Method.....	33

3.10 Stream API.....	34
3.11 The Readable Interface.....	35
3.12 Example Reading Data in Chunks.....	35
3.13 The Writable Interface.....	35
3.14 Summary.....	36
Chapter 4 - Events in Node JS.....	37
4.1 Event Driven Programming.....	37
4.2 Event Driven Programming (Contd.).....	37
4.3 Event Emitter.....	37
4.4 EventEmitter Class.....	38
4.5 EventEmitter Class – Inheritance.....	38
4.6 The Event Loop and Event Handler.....	38
4.7 Phases Overview.....	39
4.8 Event Handlers.....	39
4.9 Example (Using EventEmitter as an Object).....	40
4.10 Example (Inheriting from EventEmitter).....	40
4.11 EventEmitter Functions.....	40
4.12 Issue with 'this' Keyword in Callback Functions.....	41
4.13 Handling this Problem.....	42
4.14 Controlling Event Callbacks in the Event Loop.....	42
4.15 Summary.....	43
Chapter 5 - Asynchronous Programming with Callbacks.....	45
5.1 Synchronous and Asynchronous.....	45
5.2 Callbacks.....	45
5.3 Creating a Callback Function	46
5.4 Calling The Callback Function	46
5.5 Callback - Another Example.....	47
5.6 Issue with 'this' Keyword in Callback Functions.....	47
5.7 Handling this Problem.....	48
5.8 Handling this Problem – Method 1 (Storing in Another Variable).....	48
5.9 Handling this Problem – Method 2 (Using Bind Function).....	49
5.10 Handling this Problem – Method 3 (Using ES6 Arrow Functions).....	50
5.11 Error Handling without Callback	51
5.12 Error Handling with Callback	51
5.13 Asynchronous Callback	52
5.14 setImmediate() and nextTick().....	53
5.15 API Example.....	53
5.16 Summary.....	53
Chapter 6 - Asynchronous Programming with Promises.....	55
6.1 The Problems with Callbacks.....	55
6.2 Introduction to Promises.....	56
6.3 Requirements for Using Promises.....	56
6.4 Creating Promises Manually.....	56
6.5 Calling the Promise-based Function.....	57
6.6 Making APIs that support both callbacks and promises.....	58
6.7 Using APIs that support both callbacks and promises.....	58

6.8 Chaining then Method / Returning a Value or a Promise from then Method.....	59
6.9 Promisifying Callbacks with Bluebird.....	60
6.10 Using Bluebird.....	60
6.11 Bluebird – List of Useful Functions.....	61
6.12 Benefit of using Bluebird over ES6 for Promisification.....	62
6.13 Error Handling in Promise-based asynchronous functions.....	63
6.14 Summary.....	63
Chapter 7 - Build and Dependency Management.....	65
7.1 Introduction.....	65
7.2 Bower Package Manager.....	65
7.3 Managing Packages Using Bower.....	66
7.4 Using Bower Packages.....	66
7.5 Describing Dependency.....	67
7.6 Grunt Build Manager.....	67
7.7 Installing Grunt Components.....	68
7.8 Writing a Grunt Build Script.....	68
7.9 Running Grunt.....	69
7.10 Running the JSHint Task.....	70
7.11 Compiling 'Less' Files.....	70
7.12 Compressing CSS Files.....	71
7.13 Gulp Build Manager.....	71
7.14 Gulp vs. Grunt.....	72
7.15 Installing Gulp Components.....	72
7.16 Writing a Build Script.....	73
7.17 Running Gulp.....	73
7.18 Compiling Less Files.....	74
7.19 Summary.....	74
Chapter 8 - Basic Web Application Development.....	75
8.1 Introduction to the HTTP Module.....	75
8.2 The Request Handler Callback Function.....	76
8.3 The Server Object.....	76
8.4 Example Use of Server Object.....	76
8.5 The Request Object.....	77
8.6 The Response Object.....	77
8.7 Parsing Request Body.....	78
8.8 Serving Static Files.....	79
8.9 The HTTP Client API.....	79
8.10 Making POST/PUT/etc. Requests.....	80
8.11 Where To go from Here?.....	80
8.12 Summary.....	80
Chapter 9 - Debugging and Unit Testing.....	83
9.1 Problem Determination Options.....	83
9.2 Using console.log.....	83
9.3 Using the 'debug' Logging Package.....	84
9.4 Configure Logging.....	84
9.5 The 'Node Inspector' Debugger.....	85

9.6 Basic Usage of the Debugger.....	85
9.7 Unit Testing Node.js Applications.....	86
9.8 Getting Setup.....	86
9.9 Writing a Test Script.....	87
9.10 Running Unit Test.....	87
9.11 Testing Asynchronous Code.....	88
9.12 Using the Chai Assert API.....	89
9.13 The Chai Expect API.....	89
9.14 Summary.....	90
Chapter 10 - Introduction to Express.....	91
10.1 Introduction to Express.....	91
10.2 Basic Routing Example.....	91
10.3 Defining Routing Rules.....	92
10.4 Route Path.....	93
10.5 The Response Object.....	93
10.6 Supplying URL Parameters.....	93
10.7 Ordering of Routes.....	94
10.8 Defining Catch All Route.....	95
10.9 Full Example Web Service.....	95
10.10 Summary.....	96
Chapter 11 - Express Middleware.....	97
11.1 Introduction to Express Middleware.....	97
11.2 Writing a Middleware Function.....	97
11.3 Binding to a Path.....	98
11.4 Order of Execution.....	99
11.5 Raising Error.....	99
11.6 Handling Error.....	100
11.7 Serving Static Files.....	101
11.8 Handling POST Request Body.....	102
11.9 Enable Response Compression.....	102
11.10 Summary.....	103
Chapter 12 - Accessing MongoDB from Node.js.....	105
12.1 Getting Started.....	105
12.2 The Connection URL.....	106
12.3 Obtaining a Collection.....	106
12.4 Inserting Documents.....	106
12.5 Updating a Document.....	107
12.6 Querying for Documents.....	107
12.7 Deleting a Document.....	108
12.8 Connection Pooling.....	108
12.9 Summary.....	109
Chapter 13 - Jade Template Engine.....	111
13.1 Introduction to Jade.....	111
13.2 Using Jade.....	111
13.3 A Simple Template.....	112
13.4 Passing Data to a Template.....	113

13.5 Basic HTML Tag Rendering.....	113
13.6 Rendering Values.....	114
13.7 Conditional Rendering.....	114
13.8 Rendering a List.....	115
13.9 Layout Template.....	115
13.10 Creating a Layout Template.....	116
13.11 Creating a Content Template.....	116
13.12 Summary.....	117
Chapter 14 - Clustering and Failover.....	119
14.1 Process Management.....	119
14.2 Managing the Process Using OS Tools.....	119
14.3 Installing a Service in Windows.....	120
14.4 Create an Upstart Script in Ubuntu.....	120
14.5 Process Management Using forever.....	121
14.6 Clustering Basics.....	122
14.7 Example Clustered Application.....	122
14.8 More About Clustering.....	122
14.9 Child Process Failover.....	123
14.10 Summary.....	123
Chapter 15 - Microservices with Node.js.....	125
15.1 Microservices.....	125
15.2 Microservices with Node.js.....	125
15.3 The Express Package	126
15.4 Installing and Using Express	126
15.5 Defining Routing Rules in Express	126
15.6 Route Path.....	127
15.7 The Response Object.....	127
15.8 A Simple Web Service with Express Example	128
15.9 Composite Services.....	128
15.10 Example - Call an API Using a Promise.....	129
15.11 Using the callApi() Function.....	129
15.12 Summary.....	130
Chapter 16 - Supertest, Spy, and Nock.....	131
16.1 SuperTest.....	131
16.2 Sample Service.....	131
16.3 Test without a Testing Framework.....	131
16.4 Test with a Testing Framework.....	132
16.5 Using Promises with SuperTest.....	132
16.6 Nock.....	133
16.7 Example.....	133
16.8 Example – Request Body.....	133
16.9 Using Query String.....	134
16.10 Specifying Replies.....	134
16.11 Summary.....	135
Chapter 17 - New Features in Node.JS Version 4, 6, and 8.....	137
17.1 Node History.....	137

17.2 Node Version Policy.....	137
17.3 LTS Release Schedule.....	138
17.4 Changes in Node.js.....	138
17.5 'npm' Modules and Native Code.....	139
17.6 Node 4.x.....	139
17.7 Arrow Functions.....	140
17.8 Arrow Functions As Parameters.....	140
17.9 Using 'this' Within Arrow Functions.....	141
17.10 ES2015 Classes.....	141
17.11 Declaring Classes.....	141
17.12 Declaring Instance Methods.....	142
17.13 Accessor Methods.....	142
17.14 Static Methods.....	143
17.15 Inheritance With Classes.....	144
17.16 Generator Functions.....	144
17.17 Generator Example.....	145
17.18 Controlling Generator Execution - next(value).....	145
17.19 Controlling Generator Execution - return(value).....	146
17.20 Controlling Generator Execution - throw(exception).....	147
17.21 Generator Recursion With 'yield*'.....	147
17.22 Tail Call Optimization.....	148
17.23 'const' and 'let'.....	149
17.24 Variable Scope.....	149
17.25 Variable Scope.....	150
17.26 Shadowing Variables.....	150
17.27 Node 5.x.....	151
17.28 Spread Operator.....	151
17.29 Node 6.x.....	152
17.30 Rest Parameter.....	152
17.31 Node 7.x.....	153
17.32 Node 8.x.....	153
17.33 Summary.....	153

Chapter 1 - Introduction to Node.js

Objectives

Become familiar with Node.js concepts

- Core Node concepts
- Event-Driven, Non-Blocking I/O
- Concurrency
- "Hello, Node World!"
- Node Package Manager (NPM)

1.1 What Is Node.js?

Node's goal is to provide an easy way to build scalable network programs.

— from nodejs.org

- A platform for building high-performance JavaScript applications.
- Built on the V8 JavaScript engine from Google.
 - ◇ V8 has a Just in Time (JIT) compiler that compiles JavaScript into native opcodes for performance.
- Single-threaded approach to concurrency. Your Node.js application code always runs in exactly one thread.
 - ◇ To take advantage of multiple CPU cores you will need to run your application from multiple processes.
- Event-Driven, Non-Blocking file and network I/O. That means you don't keep the application thread waiting. This lets us process many concurrent HTTP requests using a single application thread.
- Node Package Manager (NPM) manages packages, dependencies and installs for node.
- Main web site is <https://nodejs.org/>.

1.2 Application of Node.js

- Node.js has really taken off in the server side to build web applications

and web services quickly with fairly good performance.

- ◊ Both the client and server side code is developed using JavaScript.
- ◊ Performance is generally better than PHP but perhaps not as good as Go or Java.
- Node.js is also used to write command line tools. Many of the popular tools used by the web site front end developers to manage build, dependency and testing are written in Node.js. Examples include: Gulp, Grunt, Jasmine and others.
 - ◊ As a result, you may need to learn about Node.js irrespective of how your backend is developed (e.g. Java, .NET, PHP).
- Finally, Node.js is being used to write great looking desktop applications. They mostly use the Webkit framework to build cross platform GUI applications.

1.3 Installing Node.js and NPM

■ Redhat

```
curl --silent https://rpm.nodesource.com/setup | bash -
yum -y install nodejs npm
```

■ Ubuntu and other Debian based systems:

```
sudo apt-get update
sudo apt-get install nodejs
sudo apt-get install npm
sudo ln -s /usr/bin/nodejs /usr/bin/node
```

- Windows: Download 32bit or 64bit installer from nodejs.org. NPM is included.
- Mac OSX: Download the installer package from nodejs.org. NPM is included.
- After installation verify that node is working properly by running:

```
node --version
```

1.4 "Hello, Node World!"

- Our "Hello World" application will be a web server. It will accept HTTP requests from clients and return a response.
- This is the whole application in Node! (In a file called helloworld.js.)

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

http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello, Node World!");
  response.end();
}).listen(80);
```

- Run the application like this:

```
node helloworld.js
```

```
curl http://localhost
```

1.5 How It Works

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

http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello, Node World!");
  response.end();
}).listen(80);
```

- First line indicates that we are using http.
- Next we create a server and define a callback.
 - ◇ The callback is invoked by the event loop to handle every request.
- In this simple example we are handling all requests the same way.
 - ◇ No differentiation for HTTP Method or URL.

1.6 Built on JavaScript: Benefits

- Most widely used programming language in the world!
- JavaScript is *the* language of client-side web development.
 - ◇ Same language on the server.
- JavaScript Object Notation (JSON) is the common data interchange format for JavaScript.
- Non-Blocking paradigm common in JavaScript.

1.7 Traditional Server-Side I/O Model

- Traditional Server-Side I/O (Java, C#)
 - ◇ Synchronous
 - ◇ Multi-threaded
 - ◇ Blocking
- In the Java code below, the execution thread blocks waiting for a response from the query.
 - ◇ In order to handle multiple requests, we would need to use Java Threads or the Concurrency API (or an API like framework like Servlet that uses them).

```
String query = "select FNAME, LNAME" +  
               "from CUSTOMERS" +  
               "where COUNTRY = 'USA'";  
Statement statement = connection.createStatement();  
ResultSet resultSet = statement.executeQuery(query);
```

1.8 Disadvantages of the Traditional Approach

- Correctly coding multi-threaded apps is complicated.
- Threads block (wait) for responses.
- Blocked threads cannot do work.
- Each thread has resources (like stack memory) that are unavailable while

blocked.

- Improperly implemented threading may result in *race* or *deadlock* conditions.

1.9 Event-Driven, Non-Blocking I/O

- No blocking calls to long-running functions. Examples:
 - ◇ Network I/O
 - ◇ Disk I/O
 - ◇ Database calls
- Call a function and specify a *success* callback.
- The caller does not block and, until the event loop fires the callback, the callback is not holding on to critical resources like stack memory.
- This example of a callback uses the popular jQuery library.
 - ◇ This is not a Node.js example but the browser (where jQuery runs) is also an event-driven, non-blocking I/O environment.

```
$.get('/flight-data/DL1234', function( flightData ) {  
    // handle flight data here  
    ...  
});
```

Notes

Calling a function and specifying a callback is a common JavaScript idiom. One thing that Java and C# developers often find challenging about this is that the callback function is often an *anonymous* function defined right in the call!

1.10 Concurrency

- Java app servers use a multi-threaded concurrency model.
 - ◇ The server maintains a pool of threads.
 - ◇ Each connection is handled in a new thread.
 - ◇ The thread executes (and blocks) until the request is handled.

- Node uses an event loop for concurrency.
 - ◇ The event loop runs in a single thread.
 - ◇ Of course your I/O code runs in different threads but you do not control the threading directly.
 - ◇ Simplifies the programming model.
 - ◇ Your code does not have to worry about Deadlocks or Race conditions.

1.11 Using Node Package Manager (NPM)

- You can run NPM using this syntax:

```
npm <command> <options>
```

- Basic commands
 - ◇ **help** – List available commands.
 - ◇ **install** – Install new packages by getting them from a repository.
 - ◇ **ls** – List all installed packages.
 - ◇ **update** – Packages.
 - ◇ **link** – Manage dependencies.
 - ◇ **publish** – Make your packages available publicly for reuse.

1.12 Express

- Web Framework built on Node.
- Depends on Connect.
 - ◇ Core Web Server functionality (query strings, cookies, etc.).
 - ◇ Serves static files.
- Works with multiple template formats.
 - ◇ Embedded JavaScript (EJS).
- Framework for RESTful services.

1.13 Summary

- In this module we discussed Node.js concepts:
 - ◇ Core Node concepts
 - ◇ Event-Driven I/O
 - ◇ Concurrency
 - ◇ "Hello, Node World!"
 - ◇ Node Package Manager (NPM)

Chapter 2 - Module and Dependency Management

Objectives

We will learn the following topics in this chapter:

- What is a Node.js module and why do we need them?
- How to expose functions and variables from a module.
- How to use a module from another module.
- What are core modules and how to use them.
- Learn to use the Node Package Manager (NPM).

2.1 Nature of a Node.js Project

- Most Node.js projects that you will work on can be categorized as follows:
 - ◇ Executable application – This code is executed using the node command.
 - ◇ Reusable library – This code is used by other reusable libraries and executable applications.
- A simple executable application can be written in a single JavaScript file and executed as follows:

```
node file.js
```

- Soon you will start to face a few problems:
 - ◇ A large application written in a single file will be hard to manage.
 - ◇ If you are writing a library, you need to worry about how to expose only the functions and data relevant to the user of the library and hide the rest.
- Node.js offers a simple module structure to solve these problems.

2.2 Introduction to Modules

- Node.js lets you organize your code in modules. Advantages gained are:

- ◊ Separate a large project in logical partitions with different developers working on different modules.
- ◊ Modules developed by others can be downloaded and reused easily.
- ◊ Modules can hide internal complexities and expose only functionality that are relevant to its users.
- ◊ Modules can express their dependencies on other modules. Node.js can locate and load those modules on demand.
- The module system is quite simple. Each module is made up of a single JavaScript file.
- A module exposes its functionality using **module.exports** global variable.
- A module loads another module using the **require()** global function.

2.3 A Simple Module

- The simplest way to create a module is to create a JavaScript file by the same name as the module. Example **foo.js**.

```
//Private data
var quotes = [
  "You can't be late until you show up.",
  "The secret to creativity is knowing how to hide your
sources.",
  "Nothing needs reforming as much as other people's habits"
];
//Exposed function
module.exports.getRandomQuote = function() {
  var q = quotes[
    Math.floor(Math.random()*quotes.length)];
  module.exports.last_quote = q;

  return q;
}
//Exposed data
module.exports.last_quote = null;
```

A Simple Module

Above we have created a module in `foo.js`. This module exposes a function called `getRandomQuote`. It also exposes a variable called `last_quote`. The variable called `quotes` is private and not visible from outside the module.

2.4 Using the Module

- You can load the `foo` module from another module by using its full path name without the `".js"` extension. Example:

```
var f = require("./foo");
```

- Once loaded, you access the exposed functions and variables using the module variable.

```
f.getRandomQuote();  
console.log(f.last_quote);
```

- The path name supplied to `require()` must be either absolute or relative (i.e starts with `"/"`, `"/"` or `"/"`). Otherwise, the name is interpreted as a core module.
 - ◇ A relative path is relative to the directory of the module calling `require()`. This may not be same as the folder where you are running the **node** command.
- It is a best practice to name the module variable same as the module name. Example:

```
var foo = require("./foo");  
  
foo.getRandomQuote(); //Easy to read
```

2.5 Directory Based Modules

- When a module ships with many other dependent modules or data files, a directory based module becomes useful.

- Create a directory with the same name as the module.
- Within that directory write the module code in **index.js**.
- Store any module that your module depends on in this directory.
- A directory based module is loaded using the directory name.

2.6 Example Directory Based Module

- We will create a module called **chatter**. So we create a directory called **chatter**.
- Within that directory, we create a sub-module called **speaker**. The **speaker.js** file is like this:

```
module.exports.sayIt = function() {  
    console.log("Hi there.");  
}
```

- We then write the code for the main chatter module in **index.js**.

```
var speaker = require("./speaker")  
  
module.exports.sayTooMuch = function() {  
    for (var i = 0; i < 10; ++i) {  
        speaker.sayIt();  
    }  
}
```

2.7 Using the Module

- We load the chatter module from another module using the directory name.

```
var chatter = require("/some/path/chatter");
```

- Call a public function.

```
chatter.sayTooMuch();
```

- Note that the chatter module will automatically load all other modules that it depends on (such as speaker). This makes a directory based module easy to distribute and easy to use.
- Functions exposed by the sub modules (such as sayIt()) are not available to the user of the chatter module. If you must use such a function you will need to explicitly load the sub-module using require().

2.8 Making a Module Executable

- If a module constitutes a final executable application, it can be run like this:

```
node /path/module_file.js ← For file based module
node /path/module_dir ← For directory based module
```

- In UNIX, you can convert an executable module's JavaScript file into a shell script. Lets say we have a file called **say_much** like this:

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

```
var chatter = require("/some/path/chatter");
```

```
chatter.sayTooMuch();
```

- Make the file executable.

```
chmod a+x say_much
```

- Now you can execute the JavaScript file like a script:

```
./say_much
```

Making a Module Executable

In UNIX, you can convert a JavaScript file into a shell script by adding this line as the very first line in the file.

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

When you execute this file the shell will launch the **env** program with node as the argument. In UNIX,

env is a commonly available program. It (env) will launch node and pipe the remaining contents of the file to the standard input of node. Node will interpret the script as by reading it from the standard input.

2.9 Core Modules

- A set of modules are shipped with Node.js. These are called core modules.
- Core modules are pre-compiled into binary for fast execution.
- Load a core module by name. Make sure you don't start the name with "/", "../" or "./". Example:

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

- Popular core modules:
 - ◇ **fs** – Read and write files with asynchronous support.
 - ◇ **util** – Various utilities.
 - ◇ **http** – Communicate with a web server.
 - ◇ **buffer** – Work with large chunks of data efficiently.

2.10 Loading Module from node_modules Folders

- If a module name does not start with "/", "../" or "./" then require() behaves as described below. Example:

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

- First look for a core module by that name ("chatter" here).
- If a core module cannot be found then look for a file or folder based module in the **node_modules** folder within the directory of the module calling require(). That is look for:
 - ◇ ./node_modules/chatter.js
 - ◇ ./node_modules/chatter/index.js

- If a module cannot be located then go one level up from the directory of the module calling `require()`. Look for a module within the `node_modules` folder there. That is look for:
 - ◇ `../node_modules/chatter.js`
 - ◇ `../node_modules/chatter/index.js`
- If the module is not found, keep going up a level and repeat the search until the root of the file system is reached.

2.11 Dependency Management Using NPM

- NPM lets you download modules developed by other developers from a central repository.
- Each unit of distribution is called a **package**. A package is basically a Node.js module with a **package.json** descriptor file.
- There are two ways you can install packages.
 - ◇ Locally to a module – Only that module will be able to use the installed packages.
 - ◇ Globally – Any module in the machine will be able to use the downloaded packages.
- Generally modules that are reusable code libraries are installed locally. Modules that have executable utilities are installed globally.

2.12 Installing a Package

- To locally install a package, run this from the folder of a Node.js module that will use the downloaded modules.

```
npm install package_name
```

- ◇ NPM will create the **node_modules** folder and store the downloaded packages there.
- To globally install a package run this from anywhere:

```
sudo npm install package_name -g
```

- If the package depends on other packages then those packages will also be downloaded within the `node_modules` folder of the main downloaded package. This continues down the dependency tree.
- To use a downloaded module take the usual approach:

```
var p = require("package_name");
```

Installing a Package

To install a package globally you may have to use **sudo** in UNIX.

If you are behind a corporate proxy, you will need to configure proxy server.

```
npm config set proxy http://"user:password"@proxy.company.com:1234  
npm config set https-proxy http://"user:password"@proxy.company.com:1234
```

2.13 About Global Installation

- Consider installing a module globally when one of these apply:
 - ◇ The module has executable programs. For example the `node-inspector` module offers a command called `node-debug` that can be executed.
 - ◇ Generally a reusable code module is installed locally. But if it is very large and is used by many different Node.js programs in the machine it may be better to install it globally.
- You can run the **npm root -g** command to find out the folder where global modules are installed.
- If a global module has executable command, the command shell script is stored in a directory that is already in your `PATH`.
 - ◇ In UNIX they are usually saved in `/usr/local/bin`
 - ◇ In Windows they are saved in `C:\Users\USER_NAME\AppData\Roaming\npm` folder. This folder is added to your `PATH` by Node.js installer.

2.14 Setting Up Dependency

- To manually download all packages that your module depends on can be tedious and error prone. Every developer in the team will have to repeat that process.
- A better approach is to define the dependency in the **package.json** file in your module's folder:

```
{  
  "name": "demo-app",  
  "version": "1.0.0",  
  "dependencies": {  
    "lodash": "2.4.1",  
    "preston": "*"   
  }  
}
```

- To install all dependent packages run this from your module's folder:

```
npm install
```

- The **name** and **version** fields are mandatory in package.json.
- You can generate the package.json file using **npm init** command.

2.15 Package Version Numbering Syntax

- Let's say that a package has these versions: 2.1.0, 2.0.5, 2.0.2 and 1.0.3. You can use the following version numbers when declaring a dependency.
- Latest major release:
 - ◇ * - Points to the latest major release 2.1.0.
- Latest minor release:
 - ◇ 2, 2.x or ^2.0.0 – Points to latest minor release under 2 = 2.1.0.
 - ◇ 1, 1.x or ^1.0.2 – Points to 1.0.3.
- Latest patch release:
 - ◇ 2.1, 2.1.x or ~2.1.0 – Points to latest patch release under 2.1 = 2.1.0.

- ◇ 2.0, 2.0.x or ~2.0.1 – Points to 2.0.5.
- ◇ 1.0, 1.0.x or ~1.0.2 – Points to 1.0.3
- Complete version number:
 - ◇ 2.0.2 – Points to 2.0.2 exactly.

Package Version Numbering Syntax

NPM uses semantic version. In a version number 2.1.15:

- 2 stands for a **major release** with possibly breaking changes.
- 1 stands for a **minor release** with new features but backward compatible changes.
- 15 stands for a **patch release** with mainly bug fixes.

In `package.json` you can specify a version range using the syntax described above. For example, if you depend on the `lodash` module at major version 2 then you can possibly tolerate any minor or patch releases under that major version. As a result, you can simply state:

```
"dependencies": {  
  "lodash": "2",  
  "preston": "*"  
}
```

This way you can safely update the downloaded packages by running `"npm update"`. NPM will look for the latest version of `lodash` under major version 2 and install it.

2.16 Updating Packages

- Periodically the version of a package downloaded can become obsolete as newer versions are released. For example if version required for a package is ~2.0.1 and the latest version downloaded was 2.0.2, the package will be obsolete if 2.0.5 is released.
- To find out if any of these modules have become outdated (that is newer version is available), run:

```
npm outdated
```

- Packages only become obsolete if you use a partial version number. If you use complete version number then there is no check for obsolescence.
- To bring all local packages to the latest level:

```
npm update
```

- To view all locally installed packages and their dependencies run this from your module's folder:

```
npm ls
```

2.17 Uninstalling Packages

- You can manually uninstall individual packages:

```
npm uninstall package_name  
npm uninstall package_name --g
```

- However, if you are using dependency management using package.json, you should do this instead:
 - ◇ Remove the dependency from package.json.
 - ◇ Run **npm prune** to remove unnecessary packages.

2.18 Alternate Dependency Management

- Instead of manually editing dependencies in package.json, you can let npm maintain that section of the file using the **--save** option as you install, update and uninstall packages.

```
npm install lodash --save  
npm update --save  
npm uninstall lodash --save
```

- This approach is more convenient when you are installing a new module for the first time or uninstalling a package. To update to a new major or minor version you will still have to manually update package.json.

2.19 Summary

- Modules help us break up a large application in manageable pieces. It also facilitates team development.
- Modules can contain reusable library code that can be easily shared with others.
- A module can expose only the functions and variables useful to others and hide the rest.
- You can create a file based and directory based module.
- NPM helps you download and install module from an Internet based repository.
- You can formally specify the modules your module depends on using the package.json file.

Chapter 3 - The File System Module

Objectives

We will learn these topics in this chapter:

- How to perform basic file manipulation like rename and delete.
- How to get file meta data like size.
- How to load contents of a file.
- How to write to a file.
- Low level file I/O.
- Basics of the Stream API and why we need it.

3.1 Introduction

- The **fs** core module contains API for manipulating file system and for disk I/O.

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

- It provides both synchronous and asynchronous version of most API methods. Example:

```
//Asynchronous rename
fs.rename("test.txt", "test.txt.BAK", function(err) {
  if (err === null) {
    console.log("Rename was successful.");
  } else {
    console.log(err);
  }
});
```

```
//Synchronous rename
try {
  fs.renameSync("test.txt", "test.txt.BAK");
} catch (err) {
  console.log(err);
}
```

3.2 Basic File Manipulation

- Rename a file using **rename()**. Example in previous slide.
- Remove a file using **unlink()** method.

```
fs.unlink("/path/file", function(err) {});
```

- Remove a folder using **rmdir()**.

```
fs.rmdir("/path/dir", function(err) {});
```

- Create a directory using **mkdir()**.

```
fs.mkdir("/path/dir", function(err) {});
```

3.3 Getting File/Directory Meta Data

- Use the **stat()** method.

```
fs.stat("./file.txt", function(err, stat) {  
  if (err === null) {  
    console.log("Is file: %s", stat.isFile());  
    console.log("Last accessed: %s", stat.atime);  
  }  
});
```

- The callback receives as input an error and **Stats** object.
- Key properties and methods of the Stats class:
 - ◇ **isFile(), isDirectory()** - Returns true if the path is a file or directory.
 - ◇ **size** – The size of the file in bytes.
 - ◇ **mtime** – A Date object that stores the last modification time.
 - ◇ **atime** – A Date object that stores the last access (read or write) time.

3.4 Read an Entire File

- Use **readFile(path, callback)**. Parameters are:
 - ◇ path – The name of the file to read.
 - ◇ callback – This function is called after the entire file has been read into memory.
- The callback receives two parameters:
 - ◇ error – An error object in case of error or null.
 - ◇ data – A **Buffer** object.

```
fs.readFile("test.txt", function(err, data) {  
  if (err === null) {  
    console.log("Bytes read: " + data.length);  
  } else {  
    console.log(err);  
  }  
});
```

3.5 The Buffer Class

- **Buffer** is a global class that represents a chunk of memory allocated on the heap.
- A buffer is used to read and write data from a stream like file and network socket.
- You can also read and write from the buffer memory. Example methods:
 - ◇ writeUInt8(value, offset) – Write a byte to buffer's memory at a certain offset.
 - ◇ writeUInt16LE(value, offset) – Write a 16 bit little endian unsigned integer to the buffer memory.
 - ◇ readInt32BE(offset) – Read a 32 bit big endian integer at a certain offset.
- If buffer contains text, you can try to convert it to a string:

```
var str = buff.toString("utf-8");
```

3.6 Writing to a File

- Use **writeFile(path, data, options, callback)** to write to a file.
 - ◇ **path** – The name of the file to write to.
 - ◇ **data** – A Buffer object or a string.
 - ◇ **options** – An optional object that controls the behavior of the method.
 - ◇ **callback** – A function that is called after data is written. It receives only an error parameter.

```
var data = new Buffer(4);
//Write 4 bytes to the buffer
data.writeUInt8(10, 0);
data.writeUInt8(11, 1);
data.writeUInt8(12, 2);
data.writeUInt8(13, 3);
//Write buffer to the file
fs.writeFile("test.dat", data, function(err) {
  if (err !== null) {
    console.log(err);
  }
});
```

3.7 Reading in Chunks

- To read small amounts of data at a time, first open the file in read mode using **open()** and then use **read()**.

```
fs.open("test.txt", "r", function(err, fd) {
  if (err !== null) { throw err; }
  var data = new Buffer(4);
  //Read two bytes
  fs.read(fd, data, 0, 2, 0, function(err) {
    if (err !== null) { throw err; }
    console.log("First byte: " + data.readUInt8(0));
  });
});
```


- Parameters taken by `read()` are:
 - ◇ `fd` – The file descriptor returned by `open()`.
 - ◇ `data` – A Buffer object where data will be read into.
 - ◇ `offset` – Position in the buffer where to start writing.
 - ◇ `length` – Number of bytes to read.
 - ◇ `position` – The position in the file where to read from.
 - ◇ `callback` – Called after data is read.

3.8 Writing in Chunks

- To write chunks of buffer open the file in write mode and then call **`write()`**.

```
fs.open("test.txt", "w", function(err, fd) {
  if (err !== null) { throw err; }
  var data = new Buffer(4);
  data.writeUInt8(10, 0);
  data.writeUInt8(11, 1);
  //Write two bytes
  fs.write(fd, data, 0, 2, 0, function(err, written) {
    if (err !== null) { throw err; }
    console.log("Bytes written: " + written);
  });
});
```

- The `write()` method takes these parameters:
 - ◇ `fd` – The file descriptor returned by `open()`.
 - ◇ `data` – A Buffer object that will be written from.
 - ◇ `offset` – Position in the buffer where to get data from.
 - ◇ `length` – Number of bytes to write.
 - ◇ `callback` – Called after data is written to file.

3.9 The `open()` Method

- The `open` method takes as input:

- ◇ path – An absolute or relative (to the module) path of the file.
- ◇ flags – "r" to read. "r+" to read and write. "w" to write. "a" to append.
- ◇ callback
- The callback function gets two parameters:
 - ◇ err – An error object if there was an error opening the file or null.
 - ◇ fd – A file descriptor. This can be used later to read or write from the file.
- If you open a file in write mode ("r+", "w", "a" etc.) the file is created if it doesn't exist.

3.10 Stream API

- Stream API provides a generic interface to represent a source of data that can be read from and a target of data where data can be written to.
- These interfaces are widely implemented by different modules, such as http and file system modules. This makes it possible mix sources and targets from different modules leading to applications that are both high performance and simple to write.
- Key interfaces are:
 - ◇ **Readable** – Represents a source of data. Such as a file.
 - ◇ **Writable** – Represents a target of data where data can be written. Such as a network socket.
- Example – Copy a file efficiently.

```
var source = fs.createReadStream("test.txt");  
var target = fs.createWriteStream("test.txt.BAK");  
  
source.pipe(target);
```

Stream API

The stream Readable and Writable interfaces are widely used by many modules. This leads to many interesting solutions. For example you can pipe a file Readable to a network socket Writable to write contents of the file to the socket. The resulting code is both short and highly performant.

3.11 The Readable Interface

- Fires these key events:
 - ◇ **readable** – Fired repeatedly as data becomes available to read.
 - ◇ **data** – Same as the readable event except, a chunk of data is read as Buffer and passed to the event handler callback.
 - ◇ **end** – When no more data will be available to read.
 - ◇ **close** – When the underlying data source, like file descriptor, is closed.
- Provides these key methods:
 - ◇ **read(size)** – Reads data from the readable object and returns a Buffer.
 - ◇ **pipe(writable)** – Reads data from the readable object and writes to the writable object.

3.12 Example Reading Data in Chunks

```
var source = fs.createReadStream("large_file.txt");

source.on("data", function(buffer) {
  console.log("Received bytes: " + buffer.length);
});
```

3.13 The Writable Interface

- Fires these key events:
 - ◇ **drain** – When the underlying device can be written without buffering.
- Key methods:
 - ◇ **write(buffer, callback)** – Writes the buffer and then calls the callback. Returns true if more data can be written without buffering. If you write when the underlying device can not accept data then the writes will be buffered.
 - ◇ **cork()** – Buffer all subsequent calls.
 - ◇ **uncork()** – Flush all buffered data and stops further buffering.

3.14 Summary

- The **fs** module provides both synchronous and asynchronous API for file manipulation and I/O.
- The Stream API provides a generic way to represent a source of data to read from and destination of data to write to.

Chapter 4 - Events in Node JS

Objectives

Key objectives of this chapter

- Event Driven Programming
- Event Emitter
- EventEmitter Class
- The Event Loop
- Event Handlers
- Example
- EventEmitter Functions

4.1 Event Driven Programming

- Node.js is a single-threaded application.
- Node.js can support concurrency via the concept of events and callbacks.
- Node.js uses events heavily, which is one of the reasons why Node.js is pretty fast
- When Node.js starts its server, it initiates its variables, declares functions, and then waits for events to occur

4.2 Event Driven Programming (Contd.)

4.3 Event Emitter

- Many objects in Node.js emit events
 - ◇ net.Server emits an event each time a peer connects to it
 - ◇ fs.readStream emits an event when the file is opened.
- All objects which emit events are instances of EventEmitter.

4.4 EventEmitter Class

- EventEmitter class lies in the events module
- Events module can be imported like this

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

- From 4.x onwards, the EventEmitter class is exposed by the 'events' library:

```
const EventEmitter = require('events');
```

- EventEmitter object can be acquired like this

```
var eventEmitter = new events.EventEmitter();
```

- Or, by simply using this code

```
var eventEmitter = new events();
```

4.5 EventEmitter Class – Inheritance

- The previous syntax used EventEmitter as an object. This is not a common practice.
- Best practice is to inherit from the EventEmitter class.
- Inheritance can be performed in various ways:
 - ◇ util.inherits (useful if using ES5 syntax)
 - ◇ util.inherits(BankAccount, EventEmitter);
 - ◇ extends keyword (useful if using ES6 syntax)
 - ◇ class BankAccount extends EventEmitter {
 - ◇ ...
 - ◇ }

4.6 The Event Loop and Event Handler

- In an event-driven application, there is a main loop that listens for events
- The event loop is what allows Node.js to perform non-blocking I/O operations – despite the fact that JavaScript is single-threaded – by offloading operations to the system kernel whenever possible

- When Node.js starts, it initializes the event loop, processes the provided input script which make async API calls, scheduled timers, then begins processing the event loop.

4.7 Phases Overview

- timers
 - ◇ this phase executes callbacks scheduled by `setTimeout()` and `setInterval()`
- I/O callbacks
 - ◇ executes almost all callbacks except close callbacks and `setImmediate` (More on this in Asynchronous Programming with Callback chapter)
- idle, prepare
 - ◇ only used internally
- poll
 - ◇ retrieve new I/O events
- check
 - ◇ `setImmediate` callbacks are invoked here
- close callbacks
 - ◇ e.g. `socket.on('close', ...)`

4.8 Event Handlers

- When event is triggered, an event handler function is called
- Event handling utilizes observer pattern.
- The event handler functions act as observers.
- Binding event and event handler is done like this:

```
customObject.on("eventName", eventHandler);
```

- Event can be emitted like this:

```
this.emit("eventName");
```

4.9 Example (Using EventEmitter as an Object)

```
//include events module
var events = require('events');

//instantiate EventEmitter
var eventEmitter = new events.EventEmitter();

//Create an event handler
eventEmitter.on("DataReceived", function(data) {
    console.log("Received data: " + data);
});

//Fire the event
eventEmitter.emit("DataReceived", "Hello World");
```

4.10 Example (Inheriting from EventEmitter)

```
//include modules
var EventEmitter = require('events');
var util = require('util');

//inherit custom class (ES5) from EventEmitter
util.inherits(MyEmitter, EventEmitter);
var customObject=new MyEmitter();
//Create an event handler
customObject.on("DataReceived", function(data) {
    console.log("Received data: " + data);
});

//Fire the event
this.emit("DataReceived", "Hello World");
```

4.11 EventEmitter Functions

- on(event, listener)

- ◊ Used for binding an event to an event handler function. Appends the event handler function for the specified event.
- `addListener(event, listener)`
 - ◊ Similar to the **on** function
- `removeListener(event, listener)`
 - ◊ Removes a listener from the listener array for the specified event.
- `removeAllListeners([event])`
 - ◊ Removes all listeners.
- `once(event, listener)`
 - ◊ Adds a one-time listener to the event. The listener is invoked only the next time the event is fired, after which it is removed.
- `setMaxListeners(n)`
 - ◊ By default, EventEmitters will print a warning if more than 10 listeners are added for a particular event. This function allows the limit to be increased. Use zero for unlimited.
- `listeners(event)`
 - ◊ Returns an array of listeners for the specified event
- `emit(event, [arg1], [arg2], [...])`
 - ◊ Executes each of the listeners in order with the supplied arguments. Returns true if the event had listeners, false otherwise.

4.12 Issue with 'this' Keyword in Callback Functions

- **this** is also known as **the context**
- When an event callback is executed, the value of this will usually not be the same as when triggering function is called.
- Example

```
var MyClass = function MyClass() {  
    this.fullName = '';
```

```
this.setUserName = function(firstName, lastName)
{
    // this refers to the fullName property
    // in this object
    this.fullName = firstName + " " + lastName;
    // Displays CORRECT result
    console.log(this.fullName);
}

function getUserInput(firstName, lastName, callback) {
    callback (firstName, lastName);
}

var obj = new MyClass();
getUserInput("Bob", "Smith", obj.setUserName);
//Displays blank / INCORRECT result
console.log (obj.fullName);
```

4.13 Handling this Problem

- There are various techniques for handling **this** problem
 - ◇ Store **this** in a variable
 - ◇ **bind** method
 - ◇ ES6 Arrow Functions
- All these techniques are covered in the Callbacks chapter of the course.

4.14 Controlling Event Callbacks in the Event Loop

- There are various functions which can be used to take control over callback functions order in the event loop
 - ◇ setTimeout
 - Executes a function after the specified interval. It's a one time call
 - ◇ setInterval

- Executes a function in intervals, with the specified length of the timeout between them.
- ◇ **setImmediate**
 - Queues a function behind whatever I/O event callbacks that are already in the event queue.
 - Used for implementing asynchronous callbacks.
 - It's preferred when you have recursive callbacks.
 - Covered in the callbacks chapter.
- **process.nextTick**
 - Queues a function at the head of the event queue so it executes immediately after the current function completes.
 - Used for implementing asynchronous callbacks
 - Should be avoided when you have recursive callbacks.
 - Covered in the callbacks chapter.

4.15 Summary

- Node.js supports event-driven programming, which makes it fast.
- EventEmitter class is used to create and emit events.
- Although EventEmitter can be used as an object, but best practice is to inherit from it.

Chapter 5 - Asynchronous Programming with Callbacks

Objectives

Key objectives of this chapter

- Synchronous and Asynchronous
- Callbacks
- Creating a Callback Function
- Calling the Callback Function
- Error Handling without Callback
- Error Handling with Callback
- Asynchronous Callback
- `setImmediate()` and `nextTick()`
- API Example

5.1 Synchronous and Asynchronous

- Synchronous
 - ◇ Synchronous program executes in a sequence
 - ◇ It uses blocking code, e.g. a time-consuming loop
- Asynchronous
 - ◇ Asynchronous
 - Asynchronous program doesn't execute in a sequence
 - Asynchronous program utilizes non-blocking code
 - It uses callbacks which internally uses the event loop

5.2 Callbacks

- A callback is an asynchronous equivalent for a function
- A callback function is called at the completion of a given task.
- Callback functions are passed as argument(s) to another function.

- Most Node.js API functions support asynchronous programming with callbacks.

5.3 Creating a Callback Function

- Without callback

```
var balance = 0;
function deposit(amount) {
    balance += amount;
    return balance;
}
```

- With callback

```
function deposit(amount, callback) {
    balance += amount;
    callback(balance);
}
```

5.4 Calling The Callback Function

- Method 1

```
function amountDeposited(data) {
    console.log("New balance is: " + data);
}
```

```
deposit(500, amountDeposited);
console.log("Program ended");
```

- Method 2

```
deposit(500, function(data) {
    console.log("New balance is: " + data);
});
console.log("Program ended");
```

- Output

```
New balance is: 500
Program ended
```

5.5 Callback - Another Example

- Without callback

```
var photo = downloadPhoto('http://coolcats.com/cat.gif');
console.log("Photo downloaded!");
```

- With callback

```
downloadPhoto('http://coolcats.com/cat.gif', handlePhoto);

function handlePhoto (error, photo) {
  if (error) console.error('Download error!', error);
  else console.log('Download finished', photo);
}

console.log('Download started');
```

5.6 Issue with 'this' Keyword in Callback Functions

- **this** is also known as **the context**

- When an event callback is executed, the value of this will usually not be the same as when triggering function is called.

- Example

```
var MyClass = function MyClass() {
  this.fullName = '';

  this.setUserName = function(firstName, lastName)
  {
    // this refers to the fullName property
    // in this object
    this.fullName = firstName + " " + lastName;
    // Displays CORRECT result
    console.log(this.fullName);
  }
}

function getUserInput(firstName, lastName, callback) {
  callback (firstName, lastName);
}
```

```
var obj = new MyClass();
getUserInput("Bob", "Smith", obj.setUserName);
//Displays blank / INCORRECT result
console.log (obj.fullName);
```

5.7 Handling this Problem

- There are various techniques for handling **this** problem
 - ◇ Store **this** in a variable
 - ◇ **bind** method
 - ◇ ES6 Arrow Functions

5.8 Handling this Problem – Method 1 (Storing in Another Variable)

- An easy solution is to simply create a new variable that also refers to that object. The variable can have any name, but common ones are `self` and `that`.
- Now, 'self' will be a closure variable rather than the context variable.
- Example

```
var MyClass = function MyClass() {
    this.fullName = '';
    var self = this;

    this.setUserName = function(firstName, lastName)
    {
        // this refers to the fullName property
        // in this object
        self.fullName = firstName + " " + lastName;
        //Displays CORRECT result
        console.log(self.fullName);
    }
}

function getUserInput(firstName, lastName, callback) {
```



```
        // Now save the names
        callback (firstName, lastName);
    }

var obj = new MyClass();
getUserInput("Bob", "Smith", obj.setUserName);
//Also displays CORRECT result
console.log (obj.fullName);
```

5.9 Handling this Problem – Method 2 (Using Bind Function)

- Every function has the method **bind**
- **bind** method returns a new function with `this` bound to a value.
- The function has exactly the same behavior as the one you called `.bind` on, only that `this` was set by you. No matter how or when that function is called, `this` will always refer to the passed value.
- Example

```
var MyClass = function MyClass() {
    this.fullName = '';

    this.setUserName = (function(firstName, lastName)
    {
        // this refers to the fullName property
        // in this object
        this.fullName = firstName + " " + lastName;
        // Displays CORRECT result
        console.log(this.fullName);
    }).bind(this);
}

function getUserInput(firstName, lastName, callback) {
    callback (firstName, lastName);
}

var obj = new MyClass();
getUserInput("Bob", "Smith", obj.setUserName);
```

```
//Displays CORRECT result
console.log (obj.fullName);
```

5.10 Handling this Problem – Method 3 (Using ES6 Arrow Functions)

- ECMAScript 6 introduces *arrow functions*
- Arrow functions can be thought of as lambda functions.
- They don't have their own `this` binding.
- Instead, `this` is looked up in scope just like a normal variable.
- That means you don't have to call `.bind`.
- Example

```
var MyClass = function MyClass() {
  this.fullName = '';

  this.setUserName = (firstName, lastName) =>
  {
    // this refers to the fullName property
    // in this object
    this.fullName = firstName + " " + lastName;
    // Displays CORRECT result
    console.log(this.fullName);
  }
}

function getUserInput(firstName, lastName, callback) {
  callback (firstName, lastName);
}

var obj = new MyClass();
getUserInput("Bob", "Smith", obj.setUserName);
//Displays CORRECT result
console.log (obj.fullName);
```

5.11 Error Handling without Callback

- Function definition

```
var balance = 0;

function deposit(amount) {
  if(amount <= 0)
    throw new Exception("Invalid amount!");
  balance += amount;
}
```

- Function call

```
try {
  deposit(-100);
} catch(ex) {
  console.log(ex);
}
```

5.12 Error Handling with Callback

- Function definition

```
var balance = 0;

function deposit(amount, callback) {
  var err = null;

  if(amount <= 0)
    err = "Invalid amount!";
  else
    balance += amount;
  callback(err, balance);
}
```

- Function call (Note: This function call is still synchronous)

```
deposit(-100, function(err, data) {
  if(err) {
    console.log(err);
  }
})
```

```
    }  
    else {  
        console.log(data);  
    }  
});  
console.log("Program ended");
```

5.13 Asynchronous Callback

- Function definition

```
var balance = 0;  
  
function deposit(amount, callback) {  
    var err = null;  
  
    if(amount <= 0)  
        err = "Invalid amount!";  
    else  
        balance += amount;  
  
    process.nextTick(function() {  
        callback(err, balance);  
    });
```

- Function call

```
deposit(500, function(data) {  
    console.log("New balance is: " + data);  
});  
  
console.log("Program ended");
```

- Output

```
Program ended  
New balance is: 500
```

5.14 setImmediate() and nextTick()

- The setImmediate() function schedules a callback function to be executed immediately
 - ◇ setImmediate queues the function behind whatever callbacks that are already in the event queue.
 - ◇ Used for implementing asynchronous callbacks.
 - ◇ It's preferred when you have recursive callbacks.
- Alternatively, you can call process.nextTick(function() { callback });
 - ◇ nextTick queues the function at the head of the event queue so that it executes immediately after the current callback function completes.
 - ◇ They are approximately as fast as calling a function synchronously.
 - ◇ Left unchecked, this would starve the event loop, preventing any I/O from occurring since functions are queued at the head of the event queue.

5.15 API Example

■ Synchronous

```
var data = fs.readFileSync('input.txt');  
console.log(data.toString());
```

■ Asynchronous

```
fs.readFile('input.txt', function(err, data) {  
    if(err)  
        console.log(err);  
    console.log(data);  
});
```

5.16 Summary

- Callbacks are used for implementing asynchronous functions
- Most Node.js API functions support asynchronous calls via the callback functions

- `setTimeout()` or `nextTick()` functions are used along with callbacks to implement asynchronous functions.

Chapter 6 - Asynchronous Programming with Promises

Objectives

Key objectives of this chapter

- The Problems with Callbacks
- Introduction to Promises
- Requirements for using promises
- Creating Promises manually
- Calling the Promise-based function
- Making APIs that support both callbacks and promises
- Using APIs that support both callbacks and promises
- Promisifying callbacks with Bluebird module
- Using Bluebird
- Error handling in Promise-based asynchronous functions

6.1 The Problems with Callbacks

- Asynchronous JavaScript, or JavaScript that uses callbacks, is hard to get right intuitively.
 - ◇ The code can end up with too many nested callbacks.
 - ◇ Due to nesting / indentation, the code looks like pyramid shaped

```
mymodule.getMainData(function (param) {  
  // inline callback function ...  
  
  getSomeData(param, function (someData) {  
    // another inline callback function ...  
    getMoreData(param, function (moreData) {  
      // one more inline callback function ...  
      getYetMoreData(param, function (yetMoreData) {  
        // one more inline callback function ...  
      });  
    });  
  });  
});
```

```
// etc ...  
});
```

- Callbacks are the simplest possible mechanism for asynchronous code in JavaScript, but sacrifice the control flow.

6.2 Introduction to Promises

- Promises are a way to write asynchronous code that still appears though it is executing in a top-down way
- A promise is an abstraction for asynchronous programming.
- It's an object that provides for the return value or the exception thrown by a function that has to do some asynchronous processing
- Unlike callbacks, Promises allow more better control flow
- A promise object can be passed around and anyone with access to the promise can consume it regardless if the asynchronous operation has completed or not.

6.3 Requirements for Using Promises

- Callback-based function should be defined
- Create a wrapper function which returns a promise object (available in ES6)
- The promise object calls the callback-based function
- The promise object uses **resolve** and **reject** callback functions.
- **reject** is called when there's some error.
- **resolve** is called when data is available and there are no errors

6.4 Creating Promises Manually

- Define the callback-based function

```
function deposit(amount, callback) {  
    var err = null;
```



```
    if(amount <= 0)
        err = "Invalid amount!";
    else
        balance += amount;
    callback(err, balance);
}
```

- Define a wrapper function which returns a promise object

```
function depositAsync(amount) {
    return new Promise(function(resolve, reject) {
        deposit(amount, function(err, data) {
            if(err)
                reject(err);
            else
                resolve(data);
        });
    });
}
```

6.5 Calling the Promise-based Function

- **then** method is called
- The method accepts two callback-functions as arguments
 - ◇ The first argument is utilized when data is processed successfully
 - ◇ The second argument is utilized when there's some error

```
var promise = depositAsync(500);
```

```
promise.then(
    function(data) {
        console.log("Balance is: \n" + data);
    },
    function(err) {
        console.log(err);
    }
);
```

- A better technique for executing the **then** method is to use **then(...).catch(...)**

```
promise.then(  
  function(data) {  
    console.log("Balance is: \n" + data);  
  }  
)  
.catch(function(err) {  
  console.log(err);  
});
```

- **then(...).catch(...)** makes the syntax look more like synchronous calls
 - ◇ **try {} catch(...){}**

6.6 Making APIs that support both callbacks and promises

```
function depositAsync(amount, callback) {  
  if(callback)  
    return deposit(amount, callback);  
  else  
    return new Promise(function(resolve, reject) {  
      deposit(amount, function(err, data) {  
        if(err)  
          return reject(err);  
        else  
          resolve(data);  
      });  
    });  
}
```

6.7 Using APIs that support both callbacks and promises

- If a callback is provided, it will be called with the standard callback style arguments.

```
depositAsync(500, function(err, data) {  
  if (err)  
    console.log(err);  
  else
```

```
        console.log(data);  
    });
```

- If callback is not provided, it will return a promise object

```
depositAsync(500).then(  
    function(data) {  
        console.log("Sum is: \n" + data);  
    }  
)  
.catch(function(err) {  
    console.log(err);  
});
```

6.8 Chaining then Method / Returning a Value or a Promise from then Method

- The **then** method can return a value or a promise object
- **then** method can be chained to obtain / resolve data from the previous **then** method
- In the example below, method1 and method2 returns promise objects. method3 returns a value.

```
var mainMethod = function() {  
    var promise = new Promise(function(resolve, reject){  
        resolve({data: 'mainMethod'});  
    });  
    return promise;  
};  
  
var method1 = function(data) {  
    var promise = new Promise(function(resolve, reject){  
        resolve({data: 'method 1 - completed'});  
        console.log(data.data);  
    });  
    return promise;  
};  
  
var method2 = function(data) {  
    var promise = new Promise(function(resolve, reject){
```

```
        resolve({data: 'method 2 - completed'});
        console.log(data.data);
    });
    return promise;
};

var method3 = function(data) {
    console.log(data.data);
};

mainMethod()
    .then(method1)
    .then(method2)
    .then(method3)
    .catch(function(err) {
        console.log(err);
    });

// Output
// method 1 - completed
// method 2 - completed
```

6.9 Promisifying Callbacks with Bluebird

- Implementing an asynchronous function which returns a promise object requires extra effort.
- Asynchronous functions can be generated automatically by using Bluebird module.
- Automatic generation of asynchronous functions is also called promisification.
- Other alternatives to Bluebird are Q, RSVP, ...

6.10 Using Bluebird

- Install Bluebird

```
npm install --save-dev bluebird
```

- **Import Bluebird**

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

- **Promisify a module**

```
// assuming you have var bank = require('bank');
```

```
var bank = bluebird.promisifyAll(bank);
```

- The above code will automatically generate `depositAsync` since there's a `deposit` function which makes use of a callback.

- **Overall code will look like this**

```
var bluebird = require('bluebird');  
var bank = bluebird.promisifyAll(bank);
```

```
var promise = bank.depositAsync(500);
```

```
promise.then(  
  function(data) {  
    console.log("Sum is: \n" + data);  
  })  
  .catch(function(err) {  
    console.log(err);  
  });
```

6.11 Bluebird – List of Useful Functions

- **promisify**

- ◇ Promisifies a single function

```
var readFile = Promise.promisify(require("fs").readFile);
```

- **promisifyAll**

- ◇ Promisifies a whole library / module

```
var fs = Promise.promisifyAll(require("fs"));
```

- **all**

- ◇ Useful for when you want to wait for more than one promise to complete

```
var promises = [];
```

```
for (var i = 0; i < fileNames.length; ++i) {  
    promises.push(fs.readFileAsync(fileNames[i]));  
}  
Promise.all(promises).then(function() {  
    console.log("done");  
});
```

- **map**

- ◇ Simplifies push + all into a single function

```
Promise.map(fileNames, function(fileName) {  
    // Promise.map awaits for returned promises as well.  
    return fs.readFileAsync(fileName);  
}).then(function() {  
    console.log("done");  
});
```

- **some**

- ◇ Returns a promise that is fulfilled as soon as **count** promises are fulfilled in the array

```
Promise.some([  
    ping("ns1.example.com"),  
    ping("ns2.example.com"),  
    ping("ns3.example.com"),  
    ping("ns4.example.com")  
], 2).then(function(first, second) {  
    console.log(first, second);  
});
```

- **any**

- ◇ It is like **some** method, but with 1 as the **count** value.

- **Complete API Reference**

- ◇ <http://bluebirdjs.com/docs/api-reference.html>

6.12 Benefit of using Bluebird over ES6 for Promisification

- **promisify** and **promisifyAll** make it very easy to promisify the callbacks.
 - ◇ Node v8 only supports **util.promisify**. This function isn't available in

older version.

- ◊ Bluebird supports both `promisify` and `promisifyAll` and works regardless of Node version.
- Bluebird is significantly faster than native ES6 promises in most environments
- Bluebird is more memory efficient than ES6 promises.
- Performing operations, such as iterating through an array, calling an async operation on each element is easier with Bluebird's utility functions.
- Bluebird works even when the browser doesn't support ES6.
 - ◊ This is nice if you're creating libraries that are intended to be run through 'webify' or 'webpack'
- Bluebird has a number of built-in warnings that alert you to issues that are probably wrong code. e.g. calling a function that creates a new promise inside a **then** method without returning that promise

6.13 Error Handling in Promise-based asynchronous functions

- As seen previously, error handling is done by utilizing:
 - ◊ **reject** parameter of the returned Promise object
 - ◊ The second parameter of **then** method
 - ◊ Best practice is to use the **catch** method

6.14 Summary

- Promises can be used to overcome the problems with callbacks
- Promises can be used to simplify asynchronous function calls
- Promises can be implemented manually or by using Bluebird.

Chapter 7 - Build and Dependency Management

Objectives

In this chapter we will discuss various tools used for front end web development to automate build and manage dependent packages:

- Bower
- Grunt
- Gulp

7.1 Introduction

- For a large-scale web development project you should consider using build and package management tools.
- Tools like **Grunt** or **Gulp** automate repeatable tasks like:
 - ◇ Minifying and concatenating JavaScript and CSS files.
 - ◇ Compiling SASS and Less stylesheet code.
- Tools like **Bower** and **Node Package Manager** (NPM) greatly simplify package dependency management. They can:
 - ◇ Download packages of correct version required by your application. This creates consistency across the development team.
 - ◇ Download any packages that any other package depends on. For example, Bootstrap depends on jQuery. So getting Bootstrap should also download jQuery.

7.2 Bower Package Manager

- Bower is a package manager similar to NPM but better suited for front end development.
- Unlike NPM, it keeps a flat list of dependencies. That means only one version of a package will be maintained. This is great for front end. For example, if multiple packages require jQuery only one copy of jQuery needs to be included in the web page.
- Bower depends on Node.js, NPM and Git installed.

- Bower is installed as an NPM package.

```
sudo npm install -g bower
```

- After that the **bower** command is added to your path and you can run it like this:

```
bower <command> <options>
```

7.3 Managing Packages Using Bower

- Install a package by its name in the Bower repository. Run bower from the root of your web project.

```
bower install angular  
bower install bootstrap
```

- Packages will be installed in the **bower_components** folder.
- All dependencies will also be installed. For example, installing bootstrap will also install jQuery.
- If a package is not in the Bower repository, you can install it using its Github location.

```
bower install git://github.com/alice/cool_stuff.git
```

- Uninstall a package like this:

```
bower uninstall bootstrap
```

7.4 Using Bower Packages

- You may need to process the CSS and JS files of a package before using them. For example, minify, concatenate and compile Less files if needed.
- You can also directly use the files. For example:

```
<script src="/bower_components/angular/angular.min.js">  
</script>
```

```
<link
rel="stylesheet" href=
"/bower_components/bootstrap/dist/css/bootstrap.min.css"/>
```

7.5 Describing Dependency

- Manually installing packages can be tricky when you have a large team and there are many dependencies. Everyone has to carefully install the correct versions of the required packages.
- As a best practice, describe a project's dependencies in the **bower.json** file in the project's root folder. Example:

```
{
  "name": "bower-test",
  "dependencies": {
    "bootstrap": "~3.3.6",
    "angular": "~1.4.8"
  }
}
```

- Anyone can then download the required packages by simply running:

```
bower install
```

7.6 Grunt Build Manager

- Grunt will help you automate common front end development build tasks. Such as:
 - ◇ Minify CSS and JavaScript files.
 - ◇ Concatenate CSS and JavaScript files.
 - ◇ Compile higher level stylesheets like Less and SaaS files.
- Grunt is made up of these components:
 - ◇ **Grunt CLI** – The command line interface (script). Installed globally.
 - ◇ **Core Grunt** – The actual Grunt application. Installed locally in each project. The CLI script simply calls this local copy of a project.

- ◊ **Grunt task plugins** – You install only the tasks that you need locally in a project.
- Much of Grunt is installed locally to allow you to have different versions of Grunt in the same machine.

7.7 Installing Grunt Components

- Install the CLI globally like this:

```
sudo npm install -g grunt-cli
```

- To install Grunt core and the task plugins, add them as development time dependencies of your project in **package.json**.

```
{  
  "name": "demo-app",  
  "version": "1.0.0",  
  "devDependencies": {  
    "grunt": "*",  
    "grunt-contrib-uglify": "*",  
    "grunt-contrib-concat": "*",  
    "grunt-contrib-cssmin": "*"   
  }  
}
```

- Then install them using: **npm install**

7.8 Writing a Grunt Build Script

- A Grunt build script is defined in **Gruntfile.js** in the project's root folder.
- The script is defined as a Node.js module.
- Key things done in the script are:
 - ◊ Load all required Grunt task modules.
 - ◊ Define the list of tasks that will be run and their configurations.

```
module.exports = function(grunt) {  
  //Load the "uglify" task plugin to minify JS files.
```

```
grunt.loadNpmTasks('grunt-contrib-uglify');
// Project configuration.
grunt.initConfig({
  uglify: {
    build: {
      src: 'src/**/*.js',
      dest: 'build/my-combo-code.js'
    }
  }
});
```

Writing a Grunt Build Script

A Node.js module is basically a set of functions and variables. Functionality of a module is exposed to a caller by adding the functions and variables as properties of the **module.exports** global variable. For Gruntfile we must expose a function like this:

```
module.exports = function(grunt) {
}
```

This function is called when you run the grunt command.

In the example above, we load the JavaScript uglify task. It can be used to combine and minify JavaScript code.

```
grunt.loadNpmTasks('grunt-contrib-uglify');
```

Next we specify the list of tasks that need to be run and their configurations.

```
grunt.initConfig({
  uglify: {
    build: {
      src: 'src/**/*.js',
      dest: 'build/my-combo-code.js'
    }
  }
});
```

Here we are setting up the uglify task. It will combine all JS files from the src/ folder and minify them into a single output file build/my-combo-code.js.

7.9 Running Grunt

- From the project's root folder run this to execute the **uglify** task:

```
grunt uglify
```

- If you have many tasks to run, it will be easier to define a default task that combines all tasks:

```
grunt.registerTask('default',  
  ['jshint', 'uglify', 'cssmin', 'less']);
```

- Then simply run **grunt** to execute all tasks.

7.10 Running the JSHint Task

- JSHint is a static JavaScript code analyzer that finds many problems at development time.
- It is offered by the **grunt-contrib-jshint** Node module.

```
module.exports = function(grunt) {  
  grunt.loadNpmTasks('grunt-contrib-uglify');  
  grunt.loadNpmTasks('grunt-contrib-jshint');
```

```
  grunt.initConfig({  
    jshint: {  
      all: ['Gruntfile.js', 'src/**/*.js']  
    },  
    uglify: {  
      build: {  
        src: 'src/**/*.js',  
        dest: 'build/my-combo-code.js'  
      }  
    }  
  });
```

```
  grunt.registerTask('default', ['jshint', 'uglify']);  
};
```

7.11 Compiling 'Less' Files

- The **grunt-contrib-less** module is used to compile Less files into CSS.

- Here we compile `src/my-styles.less` into `src/my-styles.css`:

```
grunt.loadNpmTasks('grunt-contrib-less');

grunt.initConfig({
  less: {
    build: {
      files: {
        "src/my-styles.css": "src/my-styles.less"
      }
    }
  }
});
```

7.12 Compressing CSS Files

- Use the **grunt-contrib-cssmin** module to compress and concatenate CSS files.
- The example below combines all CSS files from `src/` into `dist/all-styles.min.css`.

```
grunt.loadNpmTasks('grunt-contrib-cssmin');

// Project configuration.
grunt.initConfig({
  cssmin: {
    build: {
      files: {
        "dist/all-styles.min.css": ["src/**/*.css"]
      }
    }
  }
});
```

7.13 Gulp Build Manager

- Gulp is a build manager similar to Grunt and helps you automate common front end development build tasks. Such as:

- ◊ Minify CSS and JavaScript files.
- ◊ Concatenate CSS and JavaScript files.
- ◊ Compile stylesheets like Less and SaaS files.
- Gulp is made up of these components:
 - ◊ **Gulp CLI** – The command line interface (script). Installed globally.
 - ◊ **Core Gulp** – The actual Gulp application. Installed locally in each project. The CLI script simply calls this local copy of a project.
 - ◊ **Gulp task plugins** – You install only the tasks that you need locally in a project.
- Much of Gulp is installed locally to allow you to have different versions of Gulp in the same machine.

7.14 Gulp vs. Grunt

- Gulp and Grunt both have very similar goals and feature set.
- Gulp uses JavaScript code to configure the tasks. Whereas Grunt uses mostly JavaScript object to configure the tasks. Many find Gulp a bit easier to use because of it.
- Grunt is an older tool. As of 2015 Grunt still enjoys a larger community. This translates to more available skills, plugins and documentation.
- Beyond that it is not easy to recommend one system over the other.

7.15 Installing Gulp Components

- Install the CLI globally like this:

```
sudo npm install -g gulp-cli
```

- To install Gulp core and the task plugins, add them as development time dependencies of your project in **package.json**.

```
{  
  "name": "demo-app",
```



```
"version": "1.0.0",
"devDependencies": {
  "gulp": "*",
  "gulp-concat": "*",
  "gulp-less": "*",
  "gulp-uglify": "*"
}
}
```

- Then install them using: **npm install**

7.16 Writing a Build Script

- Write the build script in **gulpfile.js** in the project's root.
- A script uses a pipeline like system to connect the source files with one or more task plugins and eventually to target files.
- The example below defines a task called **minify-js** that minifies all JS files and combines them into one.

```
var gulp = require('gulp');
var uglify = require('gulp-uglify');
var concat = require('gulp-concat');

gulp.task('minify-js', function() {
  return gulp.src('src/*.js') //Source files
    .pipe(uglify()) //Task plugin 1
    .pipe(concat("all-scripts-min.js")) //Task plugin 2
    .pipe(gulp.dest('dist/')); //Destination folder
});
```

7.17 Running Gulp

- Run Gulp from the project's root like this.

```
gulp task1 task2 ...
```

- Example:

```
gulp minify-js
```

- You can define a task called **default** that is a sequence of other tasks.

```
gulp.task('default', ['minify-js', 'compile-less',  
  'minify-images']);
```

- Then simply run **gulp** to run the default task.

7.18 Compiling Less Files

- Use **gulp-less** to compile Less and **gulp-cssnano** to minimize CSS.

```
var gulp = require('gulp');  
var less = require('gulp-less');  
var cssnano = require('gulp-cssnano');  
var concat = require('gulp-concat');  
  
gulp.task('compile-less', function() {  
  return gulp.src('src/*.less') //Source files  
    .pipe(less()) //Compile Less  
    .pipe(cssnano()) //Minify CSS  
    .pipe(concat("all-styles-min.css")) //Combine  
    .pipe(gulp.dest('dist/')); //Destination folder  
});
```

7.19 Summary

- Bower is a package manager like NPM. Bower is preferred for front-end web development.
- In NPM different modules can depend on different versions of the same module. NPM will keep separate copies of the same module if needed. Bower, on the other hand, keeps only the latest version of a module. This is ideal for a web page because you will include only one version of jQuery or AngularJS in a page.
- Gulp and Grunt are build automation tools. Gulp is somewhat newer and many find it a bit easier to use.

Chapter 8 - Basic Web Application Development

Objectives

We will learn the following topics:

- How to create an HTTP server.
- How to handle HTTP requests.
- Details about the request and response objects.
- How to serve static files.

8.1 Introduction to the HTTP Module

- The **http** core module contains API to develop web server and client applications.

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

- This module is a JavaScript wrapper over the popular **libuv** C library for high performance non-blocking network I/O.
- A web server instance is created using the **createServer()** method. It takes a callback function as argument. The function is called for every HTTP request.

```
var server = http.createServer(function(req, res) {  
    res.write("Hello World\n");  
    res.end();  
});
```

- Start listening for request by calling **listen(port)**.

```
server.listen(3000);
```

Introduction to the HTTP Module

Although JavaScript has some native networking capability in the form of XMLHttpRequest (Ajax) API, Node.js does not use it. Node.js offers its own networking solution in the form of the **http** module. This module uses the **libuv** C library under the covers.

8.2 The Request Handler Callback Function

- The callback function receives two parameters
 - ◇ Request – An instance of the **http.IncomingMessage** class. It implements the **Readable** stream interface.
 - ◇ Response – An instance of **http.ServerResponse** class. It implements the **Writable** stream interface.
- From the callback function send any data back to the client using the **write()** method of the response object.
- Finally call the **end()** method on the response object to signify end of the response document.

8.3 The Server Object

- Key events fired by the server object:
 - ◇ **request** – When a new request is received. The handler callback passed to **createServer()** is automatically registered to handle this event. You can attach additional handlers to handle requests.
 - ◇ **connection** – A new TCP connection is made.
 - ◇ **close** – The server is shutting down.
- Key methods:
 - ◇ **listen(port)** – Start listening for connections.
 - ◇ **close()** - Stop listening. Shuts down the server.

8.4 Example Use of Server Object

```
var server = http.createServer(function(req, res) {  
  res.write("Hello World\n");  
  res.end();  
});
```

```
server.on("request", function(req, res) {  
  if (req.url === "/shutdown") {
```

```
        server.close() ;  
    }  
});
```

```
server.listen(3000);
```

Example Use of Server Object

In the example we attach two request handler functions. The first one is attached using the `createServer()` method. The second one is attached as a handler for the "request" event fired by the server. The second handler shuts down the server if the URL path is `"/shutdown"`. That is, to end execution of this program send a request as follows:

```
curl localhost:3000/shutdown
```

8.5 The Request Object

- Key properties:
 - ◇ **headers** – Each request header is added as a property of this object. The property name is lowercased header name.
 - ◇ **method** – The HTTP method, like GET and POST.
 - ◇ **url** – The requested path including the query string.

```
var server = http.createServer(function(req, res) {  
    console.log("Content type: %s",  
        req.headers["content-type"]); // "Content-type" header  
    console.log("Method: %s", req.method);  
    // ...  
});
```

- In addition, all methods and events supported by the Readable stream interface also apply to the request object.

8.6 The Response Object

- Key properties:
 - ◇ **statusCode** – The reply status code.

- Key methods:
 - ◇ **setHeader(name, value)** – Sets a header.
 - ◇ **removeHeader(name)** – Remove a header.
 - ◇ **write(buffer[, encoding] [,callback])** – Write a Buffer object or a string to the output stream. Default encoding is utf-8 which you can override here. The callback is called after the data is fully written to the output stream. This automatically write the headers if not already written.
 - ◇ **end** – Indicate an end of the output document.

8.7 Parsing Request Body

- Use the "data" and "end" events fired by the request object to read and parse the body. Actual parsing is done by the **querystring** core module.

```
var qs = require("querystring")

var server = http.createServer(function(req, res) {
  var body = "";

  req.on("data", function(chunk) {
    body += chunk;
  });

  req.on("end", function() {
    var post = qs.parse(body);

    console.log("Email: %s", post.email);

    res.write("All done");
    res.end();
  });
});
```

Parsing Request Body

The http module does not read and parse request body. You will need to take extra steps to process POST requests. As we have discussed, the request object implements the Readable stream interface. That means we can listen for the "data" and "end" events. We do that to accumulate the request body in

a variable. We then use the `querystring` module to parse it. The `parse()` method returns an object where each request parameter is stored as a property.

8.8 Serving Static Files

- Since the response object implements the Writable stream interface you can pipe a Readable stream to it.

```
var server = http.createServer(function(req, res) {
  res.setHeader("Content-type", "text/plain");

  var source = fs.createReadStream("test.txt");

  source.pipe(res);
});
```

8.9 The HTTP Client API

- Utilize the client API of the `http` module to call external web services from your web application.
- To get the response body you must manually accumulate the data using the "data" and "end" events.

```
http.get({
  host: 'example.com',
  path: '/'
}, function(response) {
  var body = '';
  response.on('data', function(d) {
    body += d;
  });
  response.on('end', function() {
    //All done!
    console.log(body);
  });
});
```

8.10 Making POST/PUT/etc. Requests

- Use the **http.request()** method. Host name, path etc. goes in an option object.
- Write the request body using **request.write()**.

```
var options = {  
  host: 'www.xyz.com', path: '/customer',  
  method: 'POST' //Or "PUT" etc.  
};
```

```
var req = http.request(options, function(response) {  
  //Accumulate response body as already shown before  
  //...  
});
```

```
var requestBody = '{"name": 'Daffy Duck', 'id': 10}";  
req.write(requestBody);  
req.end();
```

8.11 Where To go from Here?

- The http module provides the low level foundation for web programming. But on its own it's only good for simple applications.
- You will need a web framework for a real life application. This is where you need a framework like **Express**.

8.12 Summary

- The http core module lets you create a web server and handle HTTP requests.
- The **createServer()** method creates a server object and registers a request handler callback.
- You can attach additional request handler functions by handling the "request" event fired by the server object.
- The request and response objects implement the stream Readable and

Writable interfaces respectively.

- For a larger more real life application we will need a framework like Express.

Chapter 9 - Debugging and Unit Testing

Objectives

We will learn the following topics in this chapter:

- Different options for logging.
- Using the debug logging module.
- Using the 1.5 Node Inspector debugger.
- Unit testing using the Mocha framework.
- Using the Chai assertion API.

9.1 Problem Determination Options

- Your first line of defense should be heavy logging. This will help you diagnose problems as they are happening in production. Popular options for logging are:
 - ◇ `console.log` – Prints formatted output. Lacks timestamp.
 - ◇ `debug` package. Easy to use.
 - ◇ `winston` and `bunyan` packages. More sophisticated.
- If logs and stack traces are not helpful, you will need to debug the program using a debugger. Currently 'Node Inspector' is perhaps the best choice. It uses the debugger that's built into Chrome (recall that Node uses the V8 JavaScript engine, also from Chrome).

9.2 Using `console.log`

- Easy to use. No need to install any package.

```
var book = {name: "Obscure Destinies", price: 10.99};
```

```
console.log("Book name: %s and price: %d",  
  book.name, book.price);
```

- Format strings:
 - ◇ `%s` – For string.

- ◇ %d – For number.
- Downsides:
 - ◇ No way to turn off logging. So heavy logging will degrade performance.
 - ◇ Does not include timestamp.

9.3 Using the 'debug' Logging Package

- First install the module in your application.

```
npm install debug --save
```

- From your application obtain a logger function for your module.

```
var debug = require('debug');  
var logger = debug('my_module');
```

- Perform logging using the logger function.

```
var book = {name: "Obscure Destinies", price: 10.99};  
logger("Book name: %s and price: %d",  
      book.name, book.price);
```

9.4 Configure Logging

- By default the logger function is a no-op and no logging is performed.
- To enable logging set the **DEBUG** environment variable to a comma separated list of module names. Example:

```
DEBUG=my_module node app.js
```

- Express also uses the debug module. To enable logging for Express:

```
DEBUG=my_module,express:* node app.js
```

- In Windows set the variable separately.

```
set DEBUG=my_module,express:*  
node app.js
```

9.5 The 'Node Inspector' Debugger

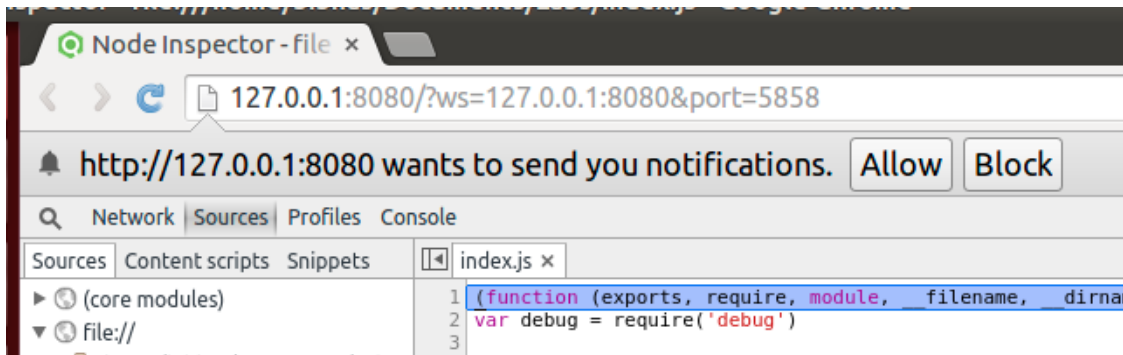
- You need to have the Chrome web browser installed. Then run:

```
sudo npm install -g node-inspector
```

- To start debugging an application:

```
node-debug index.js
```

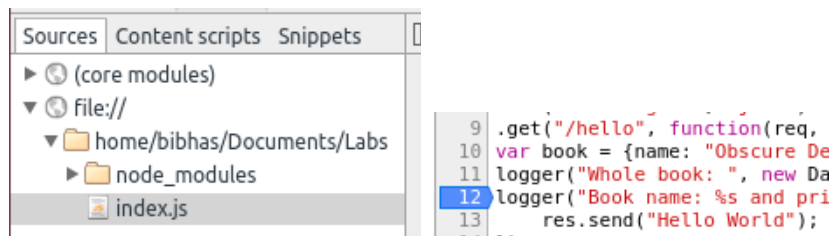
- This will launch Chrome. Allow notification for the first time.



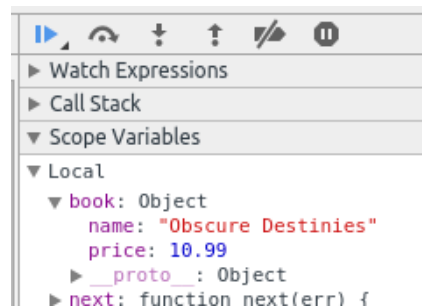
- Execution will halt at first line of the file. Click resume (▶) to continue.
- Close the browser window when done. Your program will continue to run.

9.6 Basic Usage of the Debugger

- Use the sources tab to open a source code file.



- Click left of the editor to set breakpoints.
- When stopped at a breakpoint, you can inspect variables and control execution using the right hand side pane.



9.7 Unit Testing Node.js Applications

- Unit testing is done by developers and typically finds most defects in code.
- A framework automates unit tests which encourages developers to create more tests.
- Unit test scripts can be run:
 - ◇ By developers during a build.
 - ◇ By administrators before deploying in a server.
- **Mocha** is a popular unit testing framework for Node.js.
- In addition, you can use an assertion framework that simplifies writing test scripts. These frameworks basically throw an error if a condition is not met. **Chai** is a popular assertion framework.

9.8 Getting Setup

- Dependency for mocha and chai should be setup for development only.

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

```
//Example package.json
{
  "devDependencies": {
    "mocha": "~2.2.5",
    "chai": "~3.2.0"
  },
}
```

- Configure Mocha as the test runner in package.json:

```
"scripts": {  
  "test": "mocha test"  
},
```

- Create a **test** folder in your module. This is where Mocha will look for test scripts.

9.9 Writing a Test Script

- In the **test** folder of your module create one or more JavaScript files and write test scripts there.
- A test suite is a collection of tests and is defined using the **describe()** function.
- Within a test suite define each test using the **it()** function.
- To indicate failure throw an error.

```
var my_module = require("../../my_module);
```

```
describe('Main test suite', function() {  
  it('Should make coffee', function () {  
    //All good  
  });  
  it('Should make tea', function () {  
    throw "Don't have any tea!";  
  });  
});
```

Writing a Test Script

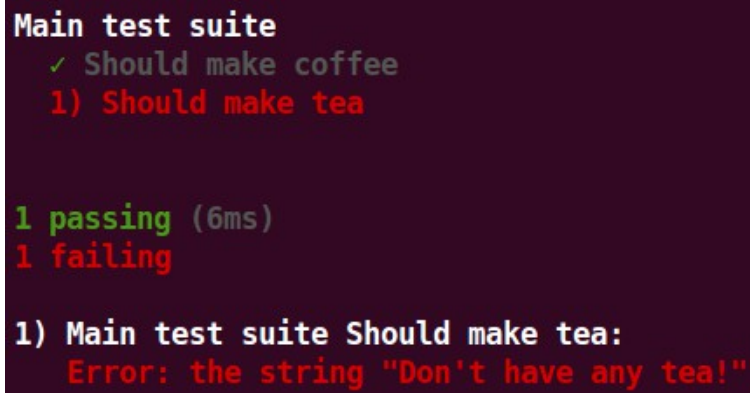
You will need to import the module that you are testing. But there is no need to import mocha. Functions like `describe()` and `it()` will be automatically made available to you from mocha.

9.10 Running Unit Test

- From the root of your module execute:

```
npm test
```

- NPM will run mocha which will provide a summary of the results.



```
Main test suite
  ✓ Should make coffee
  1) Should make tea

1 passing (6ms)
1 failing

1) Main test suite Should make tea:
     Error: the string "Don't have any tea!"
```

9.11 Testing Asynchronous Code

- Much of your code will be asynchronous. To mark successful completion of an asynchronous test, call the **done()** function passed to each test function.

```
var util = require("../my_util"); //Test my_util

describe('Main test suite', function() {
  it('Test async call', function (done) {
    //Start async work
    util.doIt(function() {
      //Call done to mark success
      done();
    });
  });
});
```

- Throw error as usual to mark failure.
- Mocha waits for a test to finish before running the next one.

Testing Asynchronous Code

Testing asynchronous code can be complex in many programming languages. Not so for Node.js if you are using Mocha. At the end of a successful completion of a test call the `done()` function passed to the test function.

Tests are run sequentially even if they are asynchronous. That is, Mocha waits for a test function to finish before running the next test. If a test function doesn't call `done()` or throw an error within a certain time, the test is considered to have failed.

9.12 Using the Chai Assert API

- The API will throw error if a condition is not met. Use it in your test scripts to test for success and failure.

```
var assert = require('chai').assert

describe('Main test suite', function() {
  it('Test assertions', function () {
    var a = 10;

    assert.equal(a, 10, "Expected 10");
    assert.notEqual(a, 20);
  });
});
```

- Other useful methods of assert.
 - ◇ `isTrue(value [,message])`
 - ◇ `isNull(value [,message]), isNotNull(value [,message])`
- Full API at: <http://chaijs.com/api/assert/>

9.13 The Chai Expect API

- This is an alternate way to perform assertion. This API uses natural language like syntax.

```
var expect = require('chai').expect

describe('Main test suite', function() {
  it('Test assertions', function () {
    var book =
      {name: "One Of Ours", price: 6.08, onSale: true};

    expect(book.onSale).to.be.true;
    expect(book.price).to.equal(6.08);
  });
});
```

```
        expect(book.price).to.be.below(20);  
    });  
});
```

- Full API at: <http://chaijs.com/api/bdd/>.

9.14 Summary

- Plan on performing extensive logging if you are developing a complex application.
- For long running applications (like a web application) it is important to add timestamp to the log entries.
- The debug package is a simple way to perform logging. It is also used by the Express framework.
- The node-inspector package lets us debug a Node.js application using the familiar Chrome development tool.
- The Mocha framework makes it easy to create automated unit test scripts.
- The Chai framework makes it easy to throw an error when a condition is not met. It can be used in conjunction with Mocha.

Chapter 10 - Introduction to Express

Objectives

Become familiar with Express concepts

- Introduction to Express
- How to define routes.
- How to handle requests.
- How to create a response document.
- How to obtain parameters from the request.

10.1 Introduction to Express

- Express is a web framework built on Node's http module.
- Provides features needed to build a large web application:
 - ◇ Routing
 - ◇ Templating
 - ◇ Static file serving.
 - ◇ Ability to plugin middleware modules that provide framework functionalities like caching, security and logging.
- Main web site is <http://expressjs.com/>.
- To install express in your module:

```
npm install express
```

- Using express:

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

10.2 Basic Routing Example

```
var express = require('express');  
var app = express(); //Create an application object
```

```
app.get('/', function (req, res) {
  res.send('Hello World!');
})
.get('/moon', function (req, res) {
  res.send('Hello Moon!');
});

app.listen(3000);
```

Basic Routing Example

In the example above we have two routing rules. The first one is for a GET request for "/". The second one is for a GET request for "/moon". This way we can have different handler functions deal with different routing rules.

10.3 Defining Routing Rules

- Each route is a unique combination of HTTP method and path.
- You define a new route using a method of the Express application object appropriate for the HTTP method. Example:
 - ◇ `get()` - For GET method.
 - ◇ `put()` - For PUT method.
 - ◇ `post()` - For POST method and so on.
- These route definition methods take a path as the first argument and a request handler function as the second argument.

```
app.put('/customer', function(req, res){});
```

- A request handler function is called when a request matches the HTTP method and path of its route. It takes two arguments:
 - ◇ Request – Same as the request object defined in the http module.
 - ◇ Response – An extension of the response object defined in the http module.

10.4 Route Path

- Path is provided as the first argument to a route definition method (get(), post() etc). It can be a string or a regular expression.
- Fully qualified path:

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

- Regular expression in a string. The following matches "/planets/earth" and "/planets/saturn".

```
app.get('/planet/*', function(req, res){});
```

- Proper regular expression. Following matches "/BMW.car", "/FORD.car" but not "/BMW.cars" or "/BMW.car/specials".

```
app.get('/.*car$/', function(req, res){});
```

10.5 The Response Object

- In addition to the methods in the response object as defined in the http module the following methods are added. All of them terminate the response document and no more data can be sent.
 - ◇ **send(data)** – Sends a string, Buffer or object. If you supply an object, it will be converted to JSON and the Content-type header will be set to "application/json; charset=utf-8". The response document is marked as completed after a call to send().
 - ◇ **redirect(path)** – Responds with a 302 redirect.

10.6 Supplying URL Parameters

- You can supply variable parameters in a URL in two ways:
 - ◇ In the path. Such as: "/planets/**earth**" and "/planets/**mars**".
 - ◇ As query string. Such as: "/planets?**sortBy=mass**".

- To use a path parameter declare it in the routing rule. Then retrieve it using **request.params** object.

```
app.get('/planets/:name', function (req, res) {  
    res.send("You asked for: " + req.params.name);  
});
```

- Retrieve a query parameter using the **request.query** object.

```
app.get('/planets', function (req, res) {  
    res.send("Sort by: " + req.query.sortBy);  
});
```

10.7 Ordering of Routes

- When a request is received Express looks for a matching route in order that they are defined.
- If a request matches multiple routes then by default only the request handler for the first matching route is executed. Below `/planets/mars` will match both routes but only the first one is executed.

```
app.get('/planets/*', function (req, res) {  
    //...  
})  
.get('/planets/:name', function (req, res) {  
    //...  
})
```

- If you must need to invoke all matching routes, you will need to call the **next** function supplied as a third argument to the request handler function.

```
app.get('/planets/*', function (req, res, next) {  
    //...  
    next(); //Invoke next matching route  
})  
.get('/planets/:name', function (req, res) {  
    //...  
})
```

10.8 Defining Catch All Route

- To route requests for a path irrespective of the HTTP method use **all()**. The following rule will be satisfied as long as the path is `"/customer"` no matter what the HTTP method is.

```
app.all('/customer', function (req, res) {  
  res.send('Matched all /\n');  
})
```

- You can define a route that will match any path and any method. Make sure it is added as the last route.

```
app.all('/*', function (req, res) {  
  res.send('Last resort');  
})
```

10.9 Full Example Web Service

```
var express = require("express");  
var app = express();  
var planets = [  
  {name: "earth", hoursInDay: 24},  
  {name: "jupiter", hoursInDay: 9.9}];  
app.get('/planets', function (req, res) {  
  res.send(planets);  
})  
.get('/planets/:name', function (req, res) {  
  var result = planets.filter(function(p) {  
    return p.name === req.params.name;  
  });  
  if (result.length === 0) {  
    res.statusCode = 404;  
    res.end();  
  } else {  
    res.send(result[0]);  
  }  
})
```

```
.listen(3000);
```

Notes

If you send a request for `http://localhost:3000/planets` this web service will return the full list of planets. If you send a request for `http://localhost:3000/planet/earth` it will only return the JSON for Earth. If you request for `http://localhost:3000/mars` you will get 404 back.

10.10 Summary

- Express is a framework to build large scale web application and web service.
- You can attach a different request handler function for different routes. Each route is a unique combination of HTTP method and URL path pattern.
- When defining a route, you can specify the path as a fully qualified string, a regular expression string and a JavaScript regular expression (RegExp object).
- A request handler function receives a request and a response object. These objects are an extension of the corresponding objects used by the core http module (discussed in a separate chapter).

Chapter 11 - Express Middleware

Objectives

We will learn the following topics in this chapter:

- What are middleware and why do we need them.
- How to develop your own middleware.
- Error handling using middleware.
- Serving static files.
- Handling POST requests.
- How to compress response body.

11.1 Introduction to Express Middleware

- A middleware is a function that gets added to the HTTP request processing pipeline.
- A middleware mainly provides infrastructural services like logging, compression of response and parsing of request body.
- A middleware can:
 - ◇ Manipulate the request and response objects.
 - ◇ Send a response to the client.
 - ◇ Terminate request processing and prevent other middlewares down the chain in the pipeline from handling the request.
- Express comes bundled with a bunch of middlewares. You can develop your own or use many middleware available from the Internet.

11.2 Writing a Middleware Function

- A middleware function takes three parameters:
 - ◇ Request – same as the request object passed to a request handler.
 - ◇ Response – same as the response object passed to a request handler.
 - ◇ Next function – This function is called to invoke the next middleware or

request handler function in the pipeline.

```
var logger = function(req, res, next) {
    console.log("Received request for: %s", req.url);
    next(); //Invoke next middleware
}
```

- A middleware function is added to the processing pipeline by calling the **use()** method of the express application object.

```
var app = express();
app.use(logger); //Add the logger middleware function
app.get("/hello", function(req, res) {
    res.send("Hello World");
});
```

11.3 Binding to a Path

- You can optionally register a middleware to a path pattern. In that case the middleware function is called only if the requested path matches that pattern.

```
app.use("/customer/:id", function(req, res, next) {
    console.log("Customer: %s", req.params.id);

    next();
});
```

- If you have multiple middlewares bound to the same path you can call **use()** multiple times or just supply all functions to a single call.

```
app.use("/customer/:id", function(req, res, next) {
    console.log("Customer: %s", req.params.id);
    next();
}, function(req, res, next) {
    console.log("Another one: %s", req.params.id);
    next();
});
```

```
});
```

11.4 Order of Execution

- Middleware functions are executed in the same order they are registered using the **use()** method.
- It is possible to call **use()** after calling a route definition method like **get()** and **post()**. But in that case, the request handler function must call **next()**.

```
app.use(function(req, res, next) {  
    console.log("First middleware");  
    next();  
});  
app.get("/hello", function(req, res, next) {  
    res.send("Hello World");  
    next(); //Request handler must call next()  
});  
app.use(function(req, res, next) {  
    console.log("Last middleware");  
    next();  
});
```

Order of Execution

The request processing pipeline is basically composed of a sequence of middleware and route handler functions (that match the request path and method). Each function must call **next()** for the next item in the list to be executed. A function can choose not to call **next()** to interrupt further request processing. For example an authorization middleware can terminate processing if user does not have enough privilege.

11.5 Raising Error

- A middleware or request handler function can raise an error in one of two ways:
 - ◇ By throwing an exception.
 - ◇ By supplying an argument to the **next()** function.
- Example:

```
app.get("/hello", function(req, res, next) {
  var invalid = true;
  //Do validation ...
  if (invalid) throw new Error("Invalid input!");

  next();
})
.use(function(req, res, next) {
  //...
  next(new Error("Something bad!"));
});
```

11.6 Handling Error

- When an error is raised all subsequent middleware and route handlers are skipped. System then looks for error handler middleware functions and executes them.
- An error handler middleware is identified by having four input parameters. The first one being the error thrown.

```
app.get("/*", function(req, res, next) {
  if (true) throw new Error("Truth hurts!");
  next();
});
app.use(function(req, res, next) {
  console.log("First middleware");
  next();
});
app.use(function(err, req, res, next) {
  console.log("An error occurred: %s", err);
  res.send("Sorry, there was a problemo");
});
```

- Error handlers must be registered after all regular middleware and route handlers.
- Error handler middleware functions are only called in case of an error.
- If you define multiple error handler middlewares, they must call the next() function.

Notes

This scheme lets you deal with all error situations occurring in your application in a single place. For example, you may want to redirect to an error page.

When an error is raised system looks for an error handler middleware down the pipeline from the point where the error was raised. This is why error handler middleware are added after all regular middleware and route definitions. Otherwise when an error is raised system will not be able to find the error handler middleware.

It is common to have multiple error handler middlewares. In that case they must all call the `next()` function. Otherwise the error handler down the pipeline will; not be called.

11.7 Serving Static Files

- Use the **express.static** built in middleware to serve static files.
- If you keep all static files within the “assets” subfolder of your application then:

```
app.use(express.static('assets'));
```

- In the example above if you request for “/images/logo.png” then the middleware will look for a file called “./assets/images/logo.png”. If found the file will be served and further processing will be skipped. Otherwise the next function in the pipeline is called.
- To avoid conflict with route handler and improve performance associate `express.static` with a path. Below, if you request for “/static/images/logo.png” then the middleware will look for a file called “./assets/images/logo.png”.

```
app.use("/static", express.static("assets"));
```

Serving Static Files

Unless you associate `express.static` with a path it will intercept all requests and try to look for a file in the designated folder (./assets in the examples above). This puts unnecessary overhead on the file system. It's a better idea to associate this middleware with a path. In this case the middleware will only look for a file if the request starts with that path.

11.8 Handling POST Request Body

- By default express does not parse a request body. You will need to use a 3rd party middleware called **body-parser**.

```
npm install body-parser
```

```
var bodyParser = require("body-parser");
```

```
app.use(bodyParser.json()); //Parse JSON body
//Parse application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({ extended: true }));
```

- This middleware will add each parameter in the body as a property of **req.body**.

```
app.post("/form", function(req, res) {
    console.log(req.body.email);
    res.send("Hello World");
});
```

```
curl --data email=daffy@wb.com localhost:3000/form
```

11.9 Enable Response Compression

- Use **compression** middleware available separately.

```
npm install compression
```

- Register it in the pipeline before any route definition and static file middleware.

```
var express = require("express");
var compress = require('compression');
var app = express();
```

```
app.use(compress())
.use(express.static('assets'))
.get("/hello", function(req, res, next) {
    res.send("Hello World");
});
```

- By default body size above 1KB will be compressed.

11.10 Summary

- A middleware is a function that is inserted into the processing pipeline to perform infrastructural work like compression, logging and authentication.
- You can write your own middleware, use the ones shipped with Node.js and download and use third party middlewares.
- For all middleware and request handler functions in the pipeline to be executed it is important that you call the `next()` function from them.
- Error handler middleware functions take four input parameters and are executed only if an error is raised.

Chapter 12 - Accessing MongoDB from Node.js

Objectives

We will learn these topics in this chapter:

- How to open connection to a MongoDB database.
- How to insert documents.
- How to update documents.
- How to perform queries.
- How to delete documents.

12.1 Getting Started

- Install the **mongodb** package in your module.

```
npm install mongodb --save
```

- Create a MongoDB client object.

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

- Open a connection to a database using a URL.

```
var url = "mongodb://localhost:27017/my_database";  
mongoClient.connect(url, function(err, db) {  
    if (err === null) {  
        console.log("Connected successfully to  
MongoDB");  
        //Use the db object  
    } else {  
        console.log(err);  
    }  
    db.close(); //Close connection when done  
});
```

- Close the connection using **close()** method of the database object.

12.2 The Connection URL

- The URL format is:
 - ◇ `mongodb://user:password@host:port/database?option=value&...`
- Where *database* is the name of the database you want to connect to.
- *user:password* is only needed if the server is password protected.
- Port number defaults to 27017 if not provided.
- Key options are:
 - ◇ `ssl=true|false|prefer` – If SSL should be used.
 - ◇ `connectTimeoutMS=val` – Connection timeout in ms.
 - ◇ `maxPoolSize=n` – Connection pool size. Defaults to 5.

12.3 Obtaining a Collection

- Insert, update, delete and query operations are done using a collection object.
- Before you can manipulate data in any way you will have to lookup the collection by name using **`db.collection()`** method. Below we look up a collection called "books".

```
mongoClient.connect(url, function(err, db) {  
    if (err === null) {  
        var col = db.collection("books");  
    }  
});
```

12.4 Inserting Documents

- Use the **`insert()`** method of the collection object to add an array of documents.

```
var col = db.collection("books");  
col.insert([
```

```
        {name: "Heart of Darkness", isbn: "014190044X"},
        {name: "Seven Pillars of Wisdom", isbn:
"1853264695"},
        {name: "The Waste Land", isbn: "0871407515"}
    ], function(err, result) {
        if (err === null) {
            console.log("Added %d books.",
result.result.n);
        }
    });
```

12.5 Updating a Document

- Use the **update()** method of the collection object to first lookup a document and then modify it.
- The following example sets the price field for a book with ISBN "014190044X".

```
var col = db.collection("books");

col.update(
    {isbn: "014190044X"}, //Lookup
    {$set: {price: 26.99}}, //Update fields
    function(err, result) {
        if (err === null) {
            console.log("Updated %d books.", result.result.n);
        }
    }
);
```

12.6 Querying for Documents

- Use the **find()** method of a collection object to do a query and obtain a **Cursor** object. Call the **toArray()** method of the cursor object to obtain an array of documents.
- Below we lookup a book by the ISBN "014190044X".

```
var col = db.collection("books");
```

```
col.find({isbn: "014190044X"})  
  .toArray(function(err, books) {  
    if (err === null) {  
      books.forEach(function(book) {  
        console.log(book.name);  
      });  
    }  
  });
```

12.7 Deleting a Document

- Lookup and remove a document by calling the `remove()` method of a collection object.
- Below we remove the book with ISBN "014190044X".

```
var col = db.collection("books");  
  
col.remove({isbn: "014190044X"}, function(err, result) {  
  if (err === null) {  
    console.log("Removed %d books.",  
result.result.n);  
  }  
});
```

12.8 Connection Pooling

- The `db` object passed to the callback by **`MongoClient.connect()`** is actually a connection pool.
- If you are using MongoDB from a web based application, open the pool once when the application starts up and keep it open for the duration of the application (that is, don't close it).

```
var app = require('express')();  
var mongodb = require("mongodb");  
var dbPool; //Connection pool  
var url = "mongodb://localhost:27017/my_database";  
  
mongodb.MongoClient.connect(url, function(err, db) {
```

```
        if (err === null) {
            dbPool = db; //Open the pool
        }
    });
    app.get('/', function(req, res) {
        //Use the connection pool here
        var col = dbPool.collection("books");
    })
    .listen(3000);
```

12.9 Summary

- You need to install the **mongodb** package to connect to MongoDB.
- A connection is opened using a URL that contains host name, port number, user id, password and database name.
- All data manipulation is done using a collection object.
- If you are using MongoDB from a web based application take care to use the connection pool correctly. Avoid creating and closing the pool for every request.

Chapter 13 - Jade Template Engine

Objectives

We will learn the following topics in this chapter:

- What is Jade?
- Installing and using Jade.
- How to render a template from an Express request handler function.
- How to pass data to a Jade template from an Express request handler.
- How to render values in Jade template.
- Conditional and iterative rendering.
- Creating layout template.

13.1 Introduction to Jade

- Jade lets you output dynamic HTML from your Express web application.
- You define the structure of a HTML document using a Jade template file. Essentially, Jade templates form the view layer of the Model View Controller pattern.
- Jade can also be used from a regular Node.js application. For example, you can generate static HTML files that embed dynamic data. This can be used for:
 - ◇ Sending HTML emails.
 - ◇ Pre-render a dynamic site as static HTML.
- Main web site: <http://jade-lang.com/>

13.2 Using Jade

- First install the module in your application.

```
npm install jade --save
```

- Create a folder within your applications root that will hold the templates.

```
mkdir views
```

- From your application instruct express to use Jade and specify the template folder.

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

```
app.set('views', __dirname + '/views')
app.set('view engine', 'jade')
```

- Normally, there is no need to load the Jade module directly from your application.

13.3 A Simple Template

- Create this as the template file **views/hello.jade**.

```
- var planet = "World"
```

```
html
  body
    p Hello #{planet}
```

- Render the template from an Express request handler.

```
app.get("/hello", function(req, res) {
  res.render("hello"); //Supply the template name
})
```

- The output HTML will be:

```
<html><body>
<p>Hello World</p>
</body></html>
```

A Simple Template

Here we have a very simple template. Within the template we have declared a JavaScript variable

called `planet` with the value "World". Embedded JavaScript code starts with `"-"` at the beginning of the line.

We render the value of the `planet` variable as HTML using `{planet}`.

The first word in a line is treated as an HTML tag (such as `html` and `body`). Indentation of the line signifies nesting. For example, `html` contains `body` and `body` contains `p`.

To load and process a template we call the `res.render()` method from an Express request handler. We pass the name of the template file minus the `.jade` extension.

13.4 Passing Data to a Template

- The real power comes from passing dynamic data from a request handler function to a template. This can be data fetched from a database or some other source.
- Data is passed as the second argument of `res.render()` method.

```
app.get("/hello", function(req, res, next) {
  res.render("hello", {
    planet: "Mars"
  });
})
```

- The object passed this way will be visible to the template.

```
html
  body
    p Hello {planet}
```

13.5 Basic HTML Tag Rendering

- The first word in a line is treated as an HTML element name.
- Indentation signifies nesting.

```
div
  p Hi there!

//Produces HTML
<div><p>Hi there!</p></div>
```

- You can supply attributes as (name="value", name="value", ...).

```
img(src="picture.jpg")
```

```
//HTML
```

```

```

- Shortcut for id and class attributes.

```
p#p1 First one
```

```
p.cool Second one
```

```
//HTML
```

```
<p id="p1">First one</p>
```

```
<p class="cool">Second one</p>
```

13.6 Rendering Values

- The **#{expression}** syntax escapes special characters.
- The **!{expression}** syntax does not escape characters.
- Use the **tag= expression** syntax when the expression makes up the whole body of a tag.

```
- var title = "Super Planets"
- var text = "I am <b>Jupiter</b>"
- var comment = "<em>All good</em>"
```

```
div= title
```

```
div !{text}
```

```
p #{comment}
```

```
//Produces HTML
```

```
<div>Super Planets</div>
```

```
<div>I am <b>Jupiter</b></div>
```

```
<p>&lt;em&gt;All good&lt;/em&gt;</p>
```

13.7 Conditional Rendering

- Use **if** and **else**. The blocks are defined using indentation and not { }.

```
div
```

```
    if auth
      p #{user}
    else
      p Please sign in

//Request handler function.

app.get("/home", function(req, res, next) {
  res.render("home", {
    auth: true,
    user: "Daffy Duck"
  });
})
```

- Be mindful of the indentation. The if statement has to be nested inside div for you to get:

```
<div><p>Daffy Duck</p></div>
```

13.8 Rendering a List

- Use **each in** to render items from a list.

```
ul
  each book in books
    li #{book}

//Request handler
.get("/book-list", function(req, res, next) {
  res.render("all_books", {
    books: [
      "Learning Python",
      "Python Pocket Reference",
      "Python in a Nutshell"
    ]
  });
})
```

13.9 Layout Template

- In a large site you should capture the layout and common elements in a

single layout template. This template has placeholders for content.

- A placeholder is defined using the **block** keyword.
- Hundreds of other templates should follow that layout and simply contribute their own content for the placeholder areas. They contribute content for a placeholder also using the **block** keyword.
- This scheme lets you change the layout of the site by changing only one file.

13.10 Creating a Layout Template

- Below is a layout template called **layout.jade**.

```
html
  head
    block header_content
  body
    p Home | Contact us | Locations

    div
      block body_content

    div This is footer text
```

13.11 Creating a Content Template

- This template uses layout.jade and provides content for the placeholders.

```
extends ../layout.jade
```

```
block header_content
  title Welcome to Home Page
```

```
block body_content
  div This is my body content
  p More stuff here.
```

- This produces:

```
<html><head>
```

```
<title>Welcome to Home Page</title>
</head><body><p>Home | Contact us |
Locations</p><div><div>This is my body content</div><p>More
stuff here.</p></div><div>This is footer text</div>
</body></html>
```

13.12 Summary

- Jade lets you output dynamic HTML from an Express based web application.
- You render a Jade template from an Express request handler using `res.render()`.
- You can pass a JavaScript object to the rendered template.
- You use the `#{expr}` `!{expr}` and `tag= expr` syntax to output the value of an expression.
- Use a layout template to simplify maintenance of a large web site.

Chapter 14 - Clustering and Failover

Objectives

We will discuss these topics in this chapter:

- How to configure a Node.js to start at machine boot time.
- How to configure failover.
- How to create a cluster for your application.

14.1 Process Management

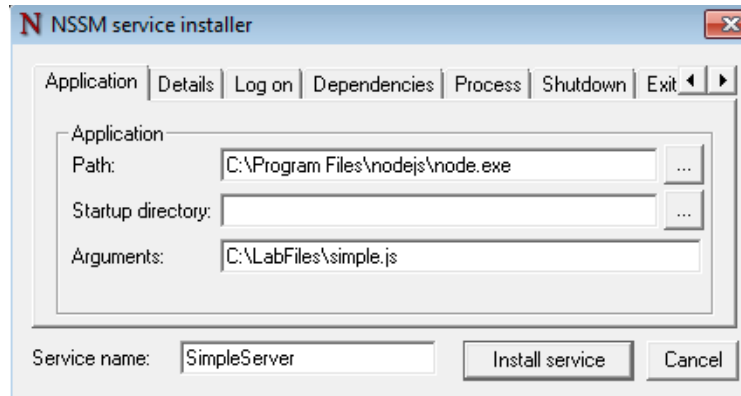
- By default a Node.js application process ends if the code throws an error.
- Two types of errors are prevalent in a Node.js application:
 - ◇ Since JavaScript is not a compiled language problems like language syntax error and invalid property names only appear at runtime.
 - ◇ Normal runtime errors such as database unavailable or an external service is unreachable.
- For a long running Node.js application such as a web server, we have a few challenges to solve:
 - ◇ The application must be launched automatically when the machine boots up.
 - ◇ If the application ends for some reason respawn it automatically.

14.2 Managing the Process Using OS Tools

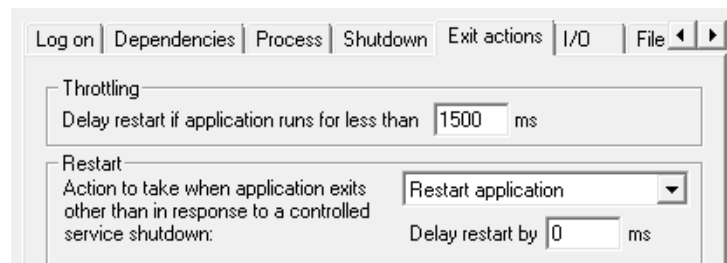
- Almost every operating system has a way to start applications on boot up and restart the process if it exits for any reason.
- In Windows, register your application as a service using the **nssm** tool.
 - ◇ Download from <http://nssm.cc/>
- RedHat 7 uses **systemd**.
- Ubuntu 14.04 uses **upstart** as the init system and migrating to systemd in near future.

14.3 Installing a Service in Windows

- Run **nssm install** command which will open a GUI.



- Provide location of Node executable and your application script.
- Enter a name for the service.



- From **Exit actions** tab make sure that **Restart application** option is selected.

14.4 Create an Upstart Script in Ubuntu

- Create a file in **/etc/init** as say **simple_server.conf**.

```
description "A simple Node.js server"
author      "Bibhas"

start on startup
stop on shutdown
respawn
script
# We found $HOME is needed.
export HOME="/home/joe"
```



```
exec sudo -u joe /usr/bin/node \  
    /home/joe/Documents/Labs/simple.js \  
    2>&1 >> /home/joe/node.log  
end script
```

- Run the service as **sudo start simple_server**.

Upstart Script

The **respawn** command tells upstart to re-run the script section if the process ends.

The file does not need any special permission.

In the example above we configure the process to run as a regular user (**joe**). This is more secure than running the process as root.

14.5 Process Management Using forever

- The **forever** Node module provides a way to respawn a Node.js application if it ends. Use it if the OS tools do not work for you for some reason.

```
sudo npm install forever -g
```

- Start your application like this:

```
forever start my_app.js
```

- View all processes being managed by forever using the **list** command. Every process has a unique ID starting with 0.

```
forever list
```

- Stop a process using its ID. Example:

```
forever stop 0
```

14.6 Clustering Basics

- By default a Node.js application code runs in a single thread. This means even under heavy load only one of the CPU cores will be busy.
- To fully utilize all CPU cores you will need to run your application from multiple processes. This is called clustering.
- Node.js comes with a core module called **cluster** which makes this very easy to setup.
 - ◇ It lets you spawn multiple child processes. Each process runs your code in its own thread.
 - ◇ Each child process listens on the same TCP port.
 - ◇ The cluster module distributes HTTP traffic load among all child processes.

14.7 Example Clustered Application

```
var http = require('http');
var cluster = require('cluster');
var numChild = require('os').cpus().length * 2;

if (cluster.isMaster) {
  //Master forks child worker processes.
  for (var i = 0; i < numChild; i++) {
    cluster.fork();
  }
} else {
  //A child process starts the actual application logic
  http.createServer(function(req, res) {
    res.writeHead(200);
    res.end("hello world\n");
  }).listen(8000);
}
```

14.8 More About Clustering

- The optimal number of child processes will depend on your application

and the number of CPU cores available. Dynamically query the number of CPU cores available and play with the formula that best suits your need. For example:

- ◊ In an N core machine allocating N-1 cores for Node.js will leave one core available for other processes. Under heavy load you will still be able to carry out administrative tasks.
- ◊ If your application blocks the main thread a lot (which is never a good idea), you may wish to set the number of child processes to a multiple of CPU cores ($n * N$).

14.9 Child Process Failover

- When a child process crashes out the parent and the other child processes keep running. The throughput will be reduced due to one less child process.
- The parent or master process keeps running as long as there is at least one child process.
- You should spawn a new child when one crashes out. This is done by handling the "exit" event from the parent process.

```
if (cluster.isMaster) {  
  for (var i = 0; i < numCPUs; i++) {  
    cluster.fork();  
  }  
  
  cluster.on("exit", function(worker, code, signal) {  
    cluster.fork(); //Replenish  
  });  
}
```

- This provides good failover. You should still use the OS tools or the forever package to setup additional failover (in case the parent process itself crashes).

14.10 Summary

- Use the tool provided by your OS to start your Node.js at machine boot

time. Use the same system to set up failover.

- Alternatively, you can use the forever package to restart failed processes.
- Clustering lets you run your application from multiple processes. This is highly recommended in multi-core machines.

Chapter 15 - Microservices with Node.js

Objectives

In this module we will discuss:

- Microservices
- Building RESTful services with Node.js

15.1 Microservices

- In the Microservices architectural style, an application is built as a suite of small services
- Each service runs in its own process
- Microservices communicate with each other using lightweight communication protocols, typically HTTP, messaging, or plain TCP/IP
- Microservices can be deployed and maintained independently of each other
- Because they typically host their own communications, they don't need an application server
 - ◇ So, they work well in a lightweight virtualization, or container-based environment

15.2 Microservices with Node.js

- Node is an excellent choice for implementing microservices:
 - ◇ All the communications protocols are either built-in or available through npm modules
 - ◇ The environment is lightweight - smallish executable, not too much memory
 - ◇ Event-Driven architecture supports high scalability
 - Thousands of concurrent connections can be supported with minimal memory overhead

15.3 The Express Package

- Express (<http://expressjs.com/>) is a web framework built around Node's *http* core module
- Very useful for rapid provisioning of RESTful web services
- Provides features needed to build large web applications, including:
 - ◇ Routing
 - ◇ Templating
 - ◇ Static file serving
 - ◇ Ability to plug in various middleware modules, like caching, security, and logging

15.4 Installing and Using Express

- To globally install the latest version Express module:

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

- Using Express in your Node apps:

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

15.5 Defining Routing Rules in Express

- Each route is a unique combination of HTTP method and path
- You define a new route using a method of the Express application object appropriate for the HTTP method. Example:
 - ◇ **get()** - For GET method
 - ◇ **put()** - For PUT method
 - ◇ **post()** - For POST method, and so on

- These route definition methods take a path as the first argument and a request (success) handler function as the second argument

```
app.post('/customer', function(req, res){});
```

- A request handler function is called when a request matches the HTTP method and path of its route. It takes two arguments:
 - ◇ **Request** – Same as the request object defined in the *http* module
 - ◇ **Response** – An extension of the response object defined in the *http* module
- Routing rules is a very important capability used in building RESTful endpoints

15.6 Route Path

- Path is provided as the first argument to a route definition method (*get()*, *post()* etc). It can be a string or a regular expression
- Fully qualified path:

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

- Regular expression in a string. The following matches *"/planets/earth"* and *"/planets/saturn"*:

```
app.get('/planet/*', function(req, res){});
```

- Proper regular expression. Following matches *"/BMW.car"*, *"/FORD.car"* but not *"/BMW.cars"* or *"/BMW.car/specials"*

```
app.get('/.*car$/', function(req, res){});
```

15.7 The Response Object

- In addition to the methods in the response object as defined in the *http*

module, the following methods are added. All of them terminate the response document and no more data can be sent

- ◇ **send(data)** – Sends a string, a buffer or an object. If you supply an object, it will be converted to JSON and the *Content-type* header will be set to *"application/json; charset=utf-8"*. The response document is marked as completed after a call to `send()`
- ◇ **redirect(path)** – Responds with an HTTP 302 redirect code

15.8 A Simple Web Service with Express Example

```
var express = require("express");
var app = express();
var planets = [
  {name: "earth", hoursInDay: 24},
  {name: "jupiter", hoursInDay: 9.9}];

app.get('/planets', function (req, res) {
  res.send(planets);
})
.get('/planets/:name', function (req, res) {
  var result = planets.filter(function(p) {
    return p.name === req.params.name;
  });

  if (result.length === 0) {
    res.statusCode = 404;
    res.end();
  } else {
    res.send(result[0]);
  }
})
.listen(3000);
```

15.9 Composite Services

- A service often needs to call one or more other services to generate a return value.
- Node's event-driven model makes it easy to launch multiple requests

15.10 Example - Call an API Using a Promise

```
const callApi = function(url) {
  return new Promise((resolve, reject) => {
    const request = http.get(url, (response) => {
      if (response.statusCode < 200 || response.statusCode > 299) {
        reject(new Error('Load Failed, status code: ' + response.statusCode + ",
url: " + url));
      }
      const body = [];
      response.on('data', (chunk) => body.push(chunk));
      response.on('end', () => {
        if (body.length > 0) {
          resolve(body.join(''));
        } else {
          resolve(null);
        }
      });
    });
    request.on('error', (err) => {
      reject(err);
    });
  });
};
```

15.11 Using the callApi() Function

- Now that we have a general function, we can call it as a function that returns a promise:

```
callApi('http://some-url.com/products')
  .then(function(value) {
    // Use the value somehow...
  }).catch(function(err) {
    // handle the error...
  });
```

- This is convenient, because we can then use Promise.all(...) to wait for multiple calls to complete...

```
var call1=callApi('http://some-url.com/products');
var call2=callApi('http://another-url.com/inventory');
Promise.all([call1, call2]).then(function(values) {
  // Do something with the values...
}).catch(function(err) {
```

```
        // handle the err...  
    });
```

15.12 Summary

- In this module we discussed Node.js concepts
 - ◇ Core Node concepts
 - ◇ Event-Driven I/O
 - ◇ Concurrency
 - ◇ Simple HTTP server with Node.js
- We also covered the basics of the Node Package Manager (NPM) and the Express package
- We provided a quick overview of the MEAN stack

Chapter 16 - Supertest, Spy, and Nock

Objectives

Key objectives of this chapter

- SuperTest
- Nock

16.1 SuperTest

- A Node.js module
- Provides a high-level abstraction for testing HTTP
- SuperTest works with and without a test framework
- Installing SuperTest

```
npm install supertest --save-dev
```

- Adding SuperTest reference

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

- Specifying URL

```
var api = supertest('<url>');
```

16.2 Sample Service

```
var express = require('express');
var app = express();
app.get('/user/1', function(req, res) {
    res.status(200).json({name: 'Bob' });
});
```

16.3 Test without a Testing Framework

- Example without any test framework

```
api.get('/user/1')
    .expect('Content-Type', 'application/json')
    .expect('Content-Length', 3)
    .expect(200)
    .end(function(err, res) {
```

```
    if(err) throw err;
  });
```

16.4 Test with a Testing Framework

- Example with a test framework

```
describe('User - tests', function() {
  it('respond with json', function(done) {
    api.get('/user/1')
      .set('Accept', 'application/json')
      .expect('Content-Type', 'application/json')
      .expect(200)
      .end(function(err, res) {
        expect(res.body).to.have.property('name');
        expect(res.body.name).to.equal('Bob');
        done();
      });
  });
});
```

- `describe()` represents a test suite and `it()` represents a test case.
- If you are using the `.end()` method `.expect()` assertions that fail will not throw exception.
- `.end()` returns the assertion as an error to the `.end()` callback
- In order to fail the test case, you will need to rethrow or pass `err` to `done()`

```
...
.end(function(err, res) {
  if(err) return done(err);
  done();
});
```

16.5 Using Promises with SuperTest

- A test case can also return a promise object

```
describe('GET /users', function() {
  it('respond with json', function() {
    return request(app)
```

```
.get('/users')
.set('Accept', 'application/json')
.expect(200)
.then(response => {
  assert(response.body.email, 'foo@bar.com')
})
});
});
```

16.6 Nock

- Nock is an HTTP mocking and expectations library for Node.js
- Nock can be used to test modules that perform HTTP requests in isolation
- e.g. if a module performs HTTP requests to a MongoDB database or makes HTTP requests to the Amazon API, you can test that module in isolation
- Installing Nock

```
npm install nock --save-dev
```

- Adding Nock reference

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

16.7 Example

```
nock('http://localhost:5000')
  .get('/users/1')
  .reply(200, {
    id: 1,
    name: 'Bob'
  });
```

- The above code will intercept every HTTP call to <http://localhost:5000>

16.8 Example – Request Body

- You can also pass request body as the second argument to get, post, put or delete specifications

- e.g.

```
nock('http://localhost:5000')
  .post('/user/2', {
    id: 2,
    name: 'Alice'
  })
  .reply(201, {
  });
```

16.9 Using Query String

- Nock understands query strings
- Instead of placing the entire URL, you can specify the query part as an object
- e.g.

```
nock('http://localhost:5000')
  .get('/users')
  .query({id: 1, name: 'Bob'})
  .reply(200, {department: 'Sales'});
```

- If the URL is <http://localhost:5000/users?id=1&name=Bob> it returns department:Sales as the json result

16.10 Specifying Replies

- Status code

```
reply(404)
```

- String

```
reply(200, 'Hello world')
```

- Number

```
reply(200, 1)
```

- JSON object

```
reply(200, {id: 1, name: 'Bob'})
```

- File

```
replyWithFile(200, __dirname + '/myfile.json')
```

- **Reply with error**

```
replyWithError('Something bad happened')
```

```
replyWithError({message: 'Something bad happened'})
```

16.11 Summary

- SuperTest provides a high-level abstraction for testing HTTP
- SuperTest works with and without a test framework
- Nock is an HTTP mocking and expectations library for Node.js
- Nock can be used to test modules that perform HTTP requests in isolation

Chapter 17 - New Features in Node.JS Version 4, 6, and 8

Objectives

Key objectives of this chapter

- New Features

17.1 Node History

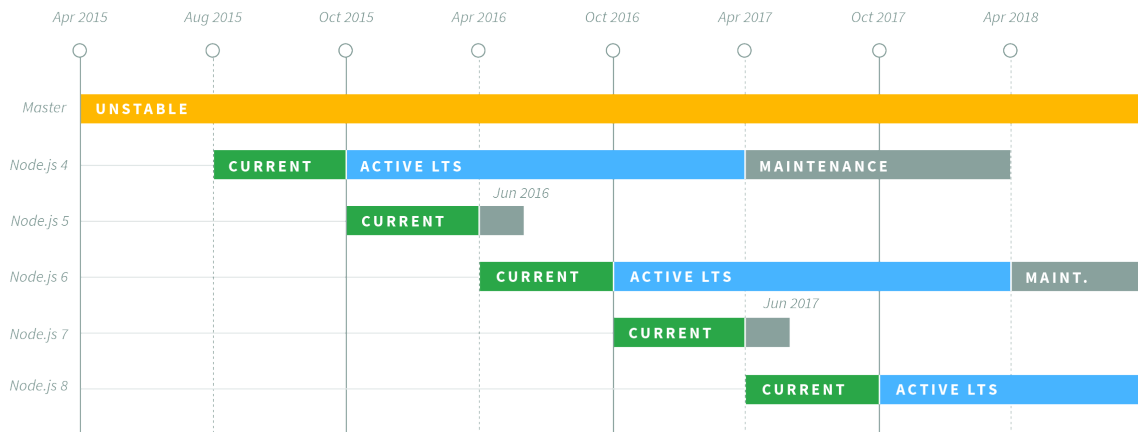
- Node was initially stewarded by a company called Joyent
- Joyent is a commercial entity (they provide cloud hosting services)
- Ryan Dahl, creator of Node, was employed by Joyent
- Node was run as an open-source project
- Joyent released Node as "0.x.x" versions up to "0.12.x"
- In 2014, there was some turmoil on the project, which led to the creation of 'io.js', which was a fork of Node.js
 - ◇ io.js released Versions "1.0" through "3.3", continuing on from the original "0.x.x" versioning scheme.
- In September 2015, Node.js v0.12 and io.js v3.3 were merged back together to become Node v4.0
- The ongoing development of Node was taken over by the new 'Node.js Foundation'

17.2 Node Version Policy

- New Major releases come out every six months
 - ◇ Even-numbered versions are cut in April
 - ◇ Odd-number versions are cut in October
- When the odd-number version is released, the previous even-numbered version enters "Long Term Support", or LTS
- LTS releases get 18 months of active support, and then 12 months of maintenance support

17.3 LTS Release Schedule

Node.js Long Term Support (LTS) Release Schedule



COPYRIGHT © 2017 NODESOURCE, LICENSED UNDER CC-BY 4.0

17.4 Changes in Node.js

- The core API (e.g. Events, File System, Net, etc) has been fairly stable since 0.12
- Each release typically includes many bug fixes and performance improvements
- The most noticeable change from version to version is the V8 JavaScript Engine version
 - ◇ This essentially means that more features from ECMAScript 2015/ES6 are available in each release
 - ◇ In the discussion that follows, we're going to only talk about "shipping" features, that are enabled by default in V8
 - ◇ The website 'node.green' provides a good overview of features by Node

version

- Detailed change logs are available at:
 - ◇ <https://github.com/nodejs/node/tree/master/doc/changelogs>

17.5 'npm' Modules and Native Code

- Many 'npm' modules use native libraries or code to implement their functionality
- As of Node 8, there is an experimental feature called "N-API" that presents a consistent programming interface for modules, independent of the V8 engine
- Until that's actually adopted...
 - ◇ You should expect to have some difficulties with native modules
 - ◇ You may find there's a bit of a lag between new Node releases and native module updates

17.6 Node 4.x

- LTS Release
 - ◇ Maintained until April 2018
- New ES2015 Features:
 - ◇ TypedArray
 - ◇ Arrow Functions
 - ◇ Classes, 'super'
 - ◇ Generators
 - ◇ 'const', 'let'
 - ◇ 'Map', 'Set' types
 - ◇ Promises

17.7 Arrow Functions

- This is a shorthand syntax for a function declaration using '=>'
- These are often used where we might need an anonymous function declaration, for example as a parameter passed to a method
- The following function defined in ES5 syntax...

```
var circleArea = function(pi, r) {  
  return pi * r * r;  
}
```

- Is the following with ES2015 arrow functions

```
var circleArea = (pi, r) => {  
  return pi * r * r;  
}
```

- ◇ And with a single statement can be simplified even further

```
var circleArea = (pi, r) => pi * r * r;
```

17.8 Arrow Functions As Parameters

- The benefits of the shorthand syntax of arrow functions is best shown when a function is a parameter of a method call and declared inline
 - ◇ These anonymous functions are declared as needed and don't need to be referenced by name
- The following example in ES5 code uses the 'Array.prototype.map()' method which returns an array of values created by executing a function passed as a parameter on each array element

```
var arr = [1, 2, 3];  
var squares = arr.map(function (x) { return x * x });
```

- The syntax in ES2015 with arrow functions is much simpler (and clearer)

```
var arr = [1, 2, 3];  
var squares = arr.map((x) => x * x);
```

- ◇ For a single parameter you can even omit the parenthesis

```
var squares = arr.map(x => x * x);
```

17.9 Using 'this' Within Arrow Functions

- In ES5, every new function defined its own **'this'** value so the following code doesn't work

```
function Person() {  
  this.age = 0;  
  setInterval(function growUp() {  
    this.age++;  
  }, 1000);  
}  
var p = new Person();
```

- In ES2015, arrow functions capture the **'this'** value of the enclosing context, so the following code works as expected

```
function Person(){  
  this.age = 0;  
  setInterval(() => {  
    this.age++; // |this| refers to the person object  
  }, 1000);  
}  
var p = new Person();
```

17.10 ES2015 Classes

- ES2015 introduced "classes" which are a new syntax for object oriented code declarations
 - ◇ This may be more "familiar" to programmers used to other programming languages
- It is important that the syntax for classes does not change the OO model of JavaScript in any way, just how code is written to use it
 - ◇ The model is still based on object constructors with prototype-based inheritance

17.11 Declaring Classes

- To declare an ES2015 class you would have two things at minimum

- ◊ The **'class'** keyword with the name of the class and a set of curly brackets for the class definition
- ◊ A function declared with the **'constructor'** keyword that will be used as the constructor
- Classes must be declared before they are referenced

```
class Media {  
  constructor(title) {  
    this.title = title;  
  }  
  
  // Other parts of class declaration  
}
```

17.12 Declaring Instance Methods

- Methods that will be invoked on individual instances of the class are simply declared within the class declaration with the method name and list of parameters
 - ◊ There is no reference to the prototype property although the method is still added to that property
 - ◊ There is also no 'function' keyword

```
class Media {  
  printTitle() {  
    console.log(this.title);  
  }  
}  
  
// equivalent to:  
// Media.prototype.printTitle = function(){  
//   console.log(this.title);  
// }
```

17.13 Accessor Methods

- If you want to have "getter" or "setter" functions called when accessing properties, you can add the **'get'** and/or **'set'** keywords in front of a

function for the name of the property

- ◊ The name of the variable used internally must be different than the name of the property being defined

```
class Media {
  constructor(title) {
    this._title = title;
  }
  get title() {
    return this._title;
  }
  set title(newTitle) {
    this._title = newTitle;
  }
}
var media1 = new Media("Kiss From a Rose");
media1.title = "Just Wanna Dance"; // set is called
console.log(media1.title); // get is called
```

Accessor Methods

The reason it is important to have the variable that stores the value of a property internally different than the property itself is to avoid recursive loops when trying to access or initialize the property.

17.14 Static Methods

- If a method is meant to be called on the class itself and not an instance, like a utility function, the keyword '**static**' is used before the method name
 - ◊ This function is not added to the prototype property
 - ◊ The function is invoked on the class itself

```
class Media {
  static countMediaInDB() {
    var count = // find how many entries
    return count;
  }
}
// equivalent to:
// Media.countMediaInDB = function() { ... };
```

```
console.log("There are " + Media.countMediaInDB()
+ " entries");
```

17.15 Inheritance With Classes

- With the new classes syntax in ES2015, inheritance is implemented with:
 - ◇ The '**extends**' keyword to setup the prototype chain between the classes
 - ◇ The '**super**' keyword to call superclass constructor or methods

```
class Song extends Media {
  constructor(title, artist) {
    super(title);
    this.artist = artist;
  }
}
class Movie extends Media {
  constructor(title, year) {
    super(title);
    this.year = year;
  }
}
```

17.16 Generator Functions

- A generator function is like a normal function, but instead of returning a single value, it returns multiple values one by one
- Calling a generator function doesn't execute its body immediately, but rather returns a new instance of the generator object
 - ◇ This generator object implements both iterable and iterator protocols
- Generator functions can be paused and resumed when the '**yield**' keyword is encountered
 - ◇ When the 'next' function of the generator object is invoked, it executes until the 'yield' keyword is encountered
 - ◇ The 'yield' keyword can be used with an expression that indicates the

value that should be returned from the call to 'next'

- ◊ The iteration is 'done' when the generator function doesn't yield any more values
- Generator functions are declared with the **'function*'** expression

17.17 Generator Example

```
function* generator_function()
{
  yield 1;
  yield 2;
  yield 3;
  yield 4;
  yield 5;
}
let generator = generator_function();
for(let i of generator) {
  console.log(i);
}
// Output:
// 1
// 2
// 3
// 4
// 5
```

17.18 Controlling Generator Execution - next(value)

- The 'next' method on a generator can take a value which is substituted as the **currently paused** 'yield' expression
 - ◊ Execution then continues until the next 'yield' pauses and optionally returns a value
 - ◊ Passing a value to the first 'next' call does nothing as the generator is not yet paused at a 'yield' expression

```
function* generator_function() {
  var a = yield 12; // line 1
```

```
    var b = yield a + 1;      // line 2
    yield b + 2;             // line 3
}
var generator = generator_function();

console.log(generator.next().value);    // 12

console.log(generator.next(5).value);   // 6

console.log(generator.next(11).value);  // 13
```

Controlling Generator Execution - next(value)

The above code executes the following way:

- The first 'next' call pauses at 'yield' in line 1 and returns 12
- The second 'next' call passes in '5' which is substituted for 'yield 12', 'a' is initialized as '5', pauses at 'yield' in line 2 and returns 6 (a + 1)
- The third 'next' call passes in '11' which is substituted for 'yield a + 1', 'b' is initialized as '11', pauses at 'yield' in line 3 and returns 13 (b + 2)

17.19 Controlling Generator Execution - return(value)

- You can anytime end a generator function before it has yielded all the values using the return() method of the generator object
- The return() method takes an optional argument, representing the final value to return
 - ◇ This value is used instead of the yield expression the generator is currently paused at

```
function* generator_function()
{
    yield 1;
    yield 2;
    yield 3;
}

var generator = generator_function();
```

```
console.log(generator.next().value);           // 1
console.log(generator.return(22).value);       // 22
```

17.20 Controlling Generator Execution - throw(exception)

- You can manually trigger an exception inside a generator function using the `throw()` method of the generator object
 - ◇ You must pass an exception to the `throw()` method that you want to throw
 - ◇ The generator function will behave as if the **currently paused 'yield'** threw the exception

```
function* genFunc1() {
  try {
    console.log('Started');
    yield;
  } catch (error) {
    console.log('Caught: ' + error);
  }
}

let genObj1 = genFunc1();
genObj1.next();
// Output: Started
genObj1.throw(new Error('Problem!'))
// Output: Caught: Error: Problem!
```

17.21 Generator Recursion With 'yield*'

- The **'yield*'** keyword inside a generator function takes an iterable object as the expression and iterates it to yield its values
 - ◇ This is used for recursive generator calls

```
function* generator_function_1() {
  yield 2;
  yield 3;
}

function* generator_function_2() {
```

```
yield 1;
yield* generator_function_1();
yield* [4, 5];
}
var generator = generator_function_2();
for(let i of generator) {
    console.log(i);
}
// Output:
// 1      2      3      4      5
```

17.22 Tail Call Optimization

- Normally when one function is called from within another function, there is a new execution stack created for the inner function call
 - ◇ This takes memory and hundreds of nested calls can exceed the call stack size of the JavaScript engine
- ES2015 defines a "tail call optimization" that avoids this nested execution stack in a very specific case, when the return of one method is exactly what is returned from calling the nested function

```
"use strict";          // only done in the "use strict" mode
function _add(x, y) {
    return x + y;
}
function add1(x, y) { ...
    //tail call
    return _add(x, y);
}
function add2(x, y) { ...
    //not tail call
    return 0 + _add(x, y);
}
```

Tail Call Optimization

The optimization avoids the creation of a new execution stack and reuses the stack of the calling outer function.

The 'add2' function above would not utilize the tail call optimization since an expression is evaluated

for the return of the outer function after the inner function is called.

Currently very few browsers or JavaScript engines actually implement the tail call optimization. This does not really impact the code though as nothing would need to be written differently, it would just start performing better when the optimization is implemented.

17.23 'const' and 'let'

- One of the bigger changes are some additional options for how to declare variables
- The **'let'** keyword declares a variable, similar to using **'var'**, that can optionally be initialized when declared
 - ◇ Unlike **'var'**, redeclaring a variable will cause an error
 - ◇ **'let'** was introduced instead of changing the behavior of **'var'**

```
var a;  
a = 13;  
var a = 24;  // legal, same as 'a = 24'
```

```
let b;  
b = 13;  
let b = 24;  // TypeError
```

- The **'const'** keyword declares a variable but it must be immediately initialized and can't change later

```
const foo;  // SyntaxError: missing = in const declaration
```

```
const bar = 123;  // OK  
bar = 456;  // TypeError: `bar` is read-only
```

17.24 Variable Scope

- One of the biggest differences between using **'var'** or **'let/const'** to declare variables is the scope of the variable
- When using **'var'**, the scope is "function scope", meaning that even if the variable is declared inside an inner block it is accessible outside

```
function myFunction() {  
    if ( ... ) {
```

```
    var a = "some text";
    console.log(a); // no problem
  }
  console.log(a); // this also works though
}
```

17.25 Variable Scope

- When using **'let'** or **'const'** the variable is at "block scope" and only available within the code block it is declared (and sub-blocks)
 - ◇ This is similar to most other languages
 - ◇ When declared outside any function they are global as expected

```
function myFunction() {
  if ( ... ) {
    let a = "some text";
    console.log(a); // no problem
  }
  console.log(a); // this gives a ReferenceError
}
```

17.26 Shadowing Variables

- Another difference between methods to declare variables is the behavior of a "shadow variable" declaration, or one that attempts to redeclare a variable in a sub-block using a variable name already in use
 - ◇ **'let'** and **'const'** will always declare a different variable
 - ◇ **'var'** will overwrite the value of the existing variable unless at the root of a function (where it would be a different variable)

```
var a = 1; // this value is never changed below
let b = 2; // this value is never changed below
```

```
function myFunction()
{
  var a = 3; //different variable
  let b = 4; //different variable
  if(true)
```

```
{
  var a = 5; //overwritten
  let b = 6; //different variable
}
// a is now 5
// b is still 4 at this scope
```

17.27 Node 5.x

- Official Support ended June 2016
- New ES2015 Features:
 - ◇ The 'spread' operator
 - ◇ 'new.target'

17.28 Spread Operator

- The **"spread operator"** splits an object that implements the new "iterable" protocol in ES2015 into the individual values
 - ◇ It is represented by the '...' token before a reference to the iterable object
- A spread operator can be placed wherever multiple function arguments or multiple elements (for array literals) are expected in code

```
function myFunction(a, b) { ... }
let data = [1, 4];
let result = myFunction(...data); // "spreads" the array
```

- A spread operator can also make array values from one array part of another array while creating it

```
let array1 = [2,3,4];
let array2 = [1, ...array1, 5, 6, 7];
console.log(array2); //Output "1, 2, 3, 4, 5, 6, 7"
```

17.29 Node 6.x

- LTS Release
 - ◇ Active support until April 2018
 - ◇ Maintained until April 2019
- Windows XP and Vista are no longer supported
- New ES2015 Features:
 - ◇ Default function parameters
 - ◇ 'rest' parameters
 - ◇ destructuring
 - ◇ Proxy
 - ◇ Reflect

17.30 Rest Parameter

- The **"rest parameter"** is used to capture a variable number of function arguments
 - ◇ It is used by adding **'...'** before the name of the last parameter in a function declaration
 - ◇ Think of it holding the **"rest"** of the parameters in an array when the number of parameters passed in the function call exceeds the number of named parameters

```
function restSum(message, ...numbers) {  
    console.log(message);  
    // add up numbers in the 'numbers' array  
}  
restSum("message 1", 1, 4, 6, 23); // array has last 4  
restSum("message 2", 2, 7); // array has last 2
```

The **'...'** operator is a rest parameter when it is before the last parameter in a function declaration but the spread operator when an existing iterable object is referenced

17.31 Node 7.x

- Expected to be unsupported in June 2017
- No major new ES2015 Features (99% compliant)

17.32 Node 8.x

- LTS Release from Oct 2017 Onwards
 - ◇ Active support until April 2019
 - ◇ Maintained until April 2020
- No major new ES2015 Features (99% compliant)
- Changes to Buffer
 - ◇ Should now be using `Buffer.alloc(...)`, not `'new Buffer(...)'`
- Experimental support for N-API
 - ◇ Stable application binary interface for native modules

17.33 Summary

- You should probably use the latest possible 'LTS' version in your production code
- Most of the changes in Node 4-8 are improvements to support for ECMAScript 2015