

UNIT II NODE JS

Basics of Node JS – Installation – Working with Node packages – Using Node package manager – Creating a simple Node.js application – Using Events – Listeners –Timers - Callbacks – Handling Data I/O – Implementing HTTP services in Node.js

Node.js

Objective : To understand the basics of Node JS and how to install it.

Course Outcome : Students are able to install and use Node js.

- Node.js is a **website/application framework** designed with high scalability in mind.
- Node.js is a great technology that is **easy to implement and yet extremely scalable**.
- Node.js is a **modular platform** (ie) much of the functionality is provided by external modules rather than being built in to the platform.
- The Node.js culture is **active in creating and publishing modules** for almost every imaginable need.
- Node.js tools are used to **build, publish, and use** your own Node.js modules in applications.

- Node.js was developed in 2009 by Ryan Dahl
- It was developed to reduce concurrency issues, especially when dealing with web services.
- Dahl created Node.js on top of V8 as a server-side environment that matched the client-side environment in the browser.
- The result is an extremely scalable server-side environment that allows developers to more easily bridge the gap between client and server.
- The fact that Node.js is written in JavaScript allows developers to easily navigate back and forth between client and server code.

Who Uses Node.js?

The following are just a few of the companies using the Node.js technology:

- Yahoo!
- LinkedIn
- eBay
- *New York Times*
- Dow Jones
- Microsoft

What Is Node.js Used For?

- Node.js can be used for a wide variety of purposes. Because it is **based on V8** and has **highly optimized code** to handle HTTP traffic.
- However, Node.js can also be used for a variety of other web services such as:
 - Web services APIs such as REST
 - Real-time multiplayer games
 - Backend web services such as cross-domain, server-side requests
 - Web-based applications
 - Multiclient communication such as IM

Installing Node.js

- To easily install Node.js, **download an installer** from the Node.js website at <http://nodejs.org>.
- The Node.js installer installs the necessary files on your PC to get Node.js up and running.
- A couple of executable files and a node_modules folder are there. The node executable file starts the Node.js JavaScript VM.

- The following list describes the executables in the Node.js install location that you need to get started:
- **node:** This file starts a Node.js JavaScript VM. If you pass in a JavaScript file location, Node.js executes that script.
- **npm:** This command is used to manage the Node.js packages.
- **node_modules:** This folder contains the installed Node.js packages. These packages act as libraries that extend the capabilities of Node.js.

Download | Node.js

https://nodejs.org/en/download

nodejs


HOME | ABOUT | DOWNLOADS | DOCS | GET INVOLVED | SECURITY


Downloads

Latest LTS Version: 18.17.0 (includes npm 9.6.7)

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

LTS
Recommended For Most Users


Windows Installer
node-v18.17.0-x64.msi


macOS Installer
node-v18.17.0.pkg

Source Code
node-v18.17.0.tar.gz

Windows Installer (.msi)

Windows Binary (.zip)

macOS Installer (.pkg)

macOS Binary (.tar.gz)

Linux Binaries (x64)

Linux Binaries (ARM)

32-bit	64-bit
32-bit	64-bit
64-bit / ARM64	
64-bit	ARM64
64-bit	
ARMv7	ARMv8

node-v18.17.0-x64.msi
Completed — 30.3 MB

Calendar(5).doc
Failed

Calendar(4).doc
Failed

medals.doc
Failed

Calendar(3).doc
Failed

Show all downloads

https://nodejs.org/dist/v18.17.0/node-v18.17.0-x64.msi

Type here to search

22°C Partly cloudy

06:03
27-07-2023

Verify Node.js Executables

- Verify that **Node.js is installed and working** before moving on. To do so, open a console prompt and execute the following command to bring up a Node.js VM:

node

- Next, at the **Node.js prompt execute the following** to write "Hello World" to the screen.

- ***>console.log("Hello World");***

- You should see "Hello World" output to the console screen. Now exit the console using **Ctrl+C** in Windows or **Cmd+C** on a Mac.

- Next, **verify that the npm command is working** by executing the following command in the OS console prompt:

npm version

You should see output similar to the following:

Click here to view code image

```
{ npm: '3.10.5',  
  ares: '1.10.1-DEV',  
  http_parser: '2.7.0',  
  icu: '57.1',  
  modules: '48',  
  node: '6.5.0',  
  openssl: '1.0.2h',  
  uv: '1.9.1',  
  v8: '5.1.281.81',  
  zlib: '1.2.8'}
```

Selecting a Node.js IDE

- To use an Integrated Development Environment (IDE) for your Node.js projects, configure that now as well.
- For example,
- Eclipse has some great Node.js plugins, and the WebStorm IDE by IntelliJ has some good features for Node.js built in.

Working with Node Packages

Objective : To work with Node packages and understand about node package manager, node package registry.

Course Outcome : Students are able to work with node packages

- One of the most **powerful features of the Node.js framework is the ability to easily extend** it with additional Node Packaged Modules (NPMs) using the Node Package Manager (NPM).

What Are Node Packaged Modules?

- A Node Packaged Module is a **packaged library** that can easily be shared, reused, and installed in different projects.
- **Many different modules are available** for a variety of purposes.
- For example, the **Mongoose module provides an ODM** (Operational Data Model) for MongoDB, **Express extends Node's HTTP capabilities**, and so on.

- Node.js modules are created by various third-party organizations to provide the needed features that Node.js lacks out of the box. This community of contributors is active in adding and updating modules.
- Node Packaged Modules include a package.json file that defines the packages.
- The package.json file includes informational metadata, such as the name, version author, and contributors, as well as control metadata, such as dependencies and other requirements that the Node Package Manager uses when performing actions such as installation and publishing.


Understanding the Node Package Registry

- The Node modules have a managed location called the Node Package Registry where packages are registered. This allows you to publish your own packages in a location where others can use them as well as download packages that others have created.
- The Node Package Registry is located at <https://npmjs.com>. From this location you can view the newest and most popular modules as well as search for specific packages, as shown in Figure 3.1.


npm

< > ⏮ ⏭ ↺ 🏠 NPM, Inc. [US] https://www.npmjs.com/ 🔍 Search Google


Packages people 'npm install' a lot




browserify
browser-side require() the node way
14.4.0 published 2 months ago by **feross**




grunt-cli
The grunt command line interface
1.2.0 published a year ago by **vladikoff**




bower
The browser package manager
1.8.0 published 8 months ago by **sheerun**



gulp
The streaming build system
3.9.1 published a year ago by **phated**



grunt
The JavaScript Task Runner
1.0.1 published a year ago by **shama**



express
Fast, unopinionated, minimalist web fram...
4.15.3 published 2 months ago by **dougwilson**

Using the Node Package Manager

- The Node Package Manager you have already seen is a command-line utility.
- It allows you to **find, install, remove, publish**, and do everything else related to Node Package Modules.
- The Node Package Manager **provides the link between the Node Package Registry and your development environment.**

**npm command
options
(with express
as the package
where appro**

<code>install</code>	Installs a package either using a <u>package.json</u> file, from the repository, or a local location	<code><u>npm</u> install</code> <code><u>npm</u> install express</code> <code>npm install express@0.1.1</code> <code>npm install ../tModule.tgz</code>
<code>install -g</code>	Installs a package globally	<code>npm install express -g</code>
<code>remove</code>	Removes a module	<code>npm remove express</code>
<code>pack</code>	Packages the module defined by the <u>package.json</u> file into a .tgz file	<code>npm pack</code>

view	Displays module details	npm view express
publish	Publishes the module defined by a package.json file to the registry	npm publish
unpublish	Unpublishes a module you have published	npm unpublish myModule
owner	Allows you to add, remove, and list owners of a package in the repository	npm add bdayley myModule npm rm bdayley myModule npm ls myModule

- **Searching for Node Package Modules**

- You can also search for modules in the Node Package Registry directly from the command prompt using the `npm search <search_string>` command.
- For example, the following command searches for modules related to openssl and displays the results :

npm search openssl

Installing Node Packaged Modules

- To install a Node module, use the `npm install <module_name>` command.
- This downloads the Node module to your development environment and places it into the `node_modules` folder where the install command is run.
- For example, the following command installs the `express` module:

npm install express

C:\express\example

`-- [express@4.14.0](#)

+-- [accepts@1.3.3](#)

| +-- [mime-types@2.1.11](#)

| | `-- [mime-db@1.23.0](#)

| `-- [negotiator@0.6.1](#)

+-- [array-flatten@1.1.1](#)

+-- [content-disposition@0.5.1](#)

+-- [content-type@1.0.2](#)

+-- [cookie@0.3.1](#)

+-- [cookie-signature@1.0.6](#)

+-- [debug@2.2.0](#)

| `-- [ms@0.7.1](#) ...

The dependency hierarchy is listed; some of the methods Express requires are cookie-signature, range-parser, debug, fresh, cookie, and send modules.

- Node.js has to be able to **handle dependency conflicts**.
- For example, the express module requires ***cookie 0.3.1***, but another module may require ***cookie 0.3.0***.
- To handle this situation, a separate copy for the cookie module is placed in each module's folder under another ***node_modules*** folder.

- To illustrate how modules are stored in a hierarchy, consider the following example of how express looks on disk.

./
./node_modules
./node_modules/express
./node_modules/express/node_modules/cookie
./node_modules/express/node_modules/send
./node_modules/express/node_modules/send/node_modules/mime

- **Using package.json**

- All Node modules must include a **package.json** file in their **root directory**.
- The **package.json** file is a **simple JSON text file** that defines the module including dependencies.
- The **package.json** file can contain a **number of different directives** to tell the Node Package Manager how to handle the module.

```
{  
  "name": "my_module",  
  "version": "0.1.0",  
  "description": "a simple node.js module",  
  "dependencies" : {"express" : "latest"}  
}
```

The only required directives in the package.json file are name and version.

Table 3.2 Directives used in the package.json file

Directive	Description	Example
name	Unique name of package.	"name": " <u>camelot</u> "
<u>preferGlobal</u>	Indicates this module prefers to be installed globally.	" <u>preferGlobal</u> ": true
version	Version of the module.	"version": 0.0.1
author	Author of the project.	"author": " <u>arthur@???.com</u> "
description	Textual description of module.	"description": "a silly place"
<u>contributors</u>	Additional contributors to the module.	"contributors": [{ " <u>name</u> ": " <u>gwen</u> ", " <u>email</u> ": " <u>gwen@???.com</u> " }]
bin	Binary to be installed globally with project.	"bin": { " <u>excalibur</u> ": " <u>./bin/excalibur</u> "}
scripts	Specifies parameters that execute console apps	"scripts" { "start": " <u>node</u> <u>./bin/</u>

	when launching node.	<code>excalibur", "test": "echo testing"}</code>
main	Specifies the main entry point for the app. This can be a binary or a <code>.js</code> file.	<code>"main": "./bin/excalibur"</code>
repository	Specifies the repository type and location of the package.	<code>"repository": { "type": "git", "location": "http://????.com/c_git"}</code>
keywords	Specifies keywords that show up in the <code>npm</code> search.	<code>"keywords": ["swallow", "unladen"]</code>
dependencies	Modules and versions this module depends on. You can use the <code>*</code> and <code>x</code> wildcards.	<code>"dependencies": { "express": "latest", "connect": "2.x.x", "cookies": "*"}</code>
engines	Version of node this package works with.	<code>"engines": { "node": ">=6.5"}</code>

Creating a Node.js Packaged Module

Objective : To create a node.js packaged module.

Course Outcome : Students are able to create their own modules and use them.

To create a Node.js Packaged Module,

- need to **create the functionality** in JavaScript,
- **define the package** using a package.json file,
- either **publish it to the registry or package it for local use**.

Building a Node.js Packaged Module using an example called censorify.

- The censorify module accepts text and then replaces certain words with asterisks:
 - Create a project folder named .../censorify. This is the root of the package.
 - Inside that folder, create a file named censortext.js.
 - Add the below code to censortext.js.

- A great way to use package.json files is to automatically download and install the dependencies for your Node.js app
- All you need to do is create a package.json file in the root of your project code and add the necessary dependencies to it.
- For example, the following package.json requires the express module as a dependency.

```
{  
  "name": "my_module",  
  "version": "0.1.0", "dependencies" : {  
    "express" : "latest"  
  }  
}
```

Then you run the following command from root of your package, and the express module is automatically installed.

- **Implementing a simple censor function and exporting it for other modules using the package**

```
var censoredWords = ["sad", "bad", "mad"];
var customCensoredWords = [];
function censor(inStr)
{
    for (idx in censoredWords)
    {
        inStr = inStr.replace(censoredWords[idx], "*****");
    }
    for (idx in customCensoredWords)
    {
        inStr = inStr.replace(customCensoredWords[idx], "*****");
    }
    return inStr;
}
```

```
function addCensoredWord(word)
{
    customCensoredWords.push(word);
}
function getCensoredWords()
{
    return censoredWords.concat(customCensoredWords);
}
exports.censor = censor;
exports.addCensoredWord = addCensoredWord;
exports.getCensoredWords = getCensoredWords;
```


Once the module code is completed, you need to **create a `package.json`** file that is used to generate the Node.js Packaged Module.

Create a `package.json` file in the `../censorify` folder.

Specifically, you need to **add the name, version, and main directives** as a minimum.

The main directive needs to be the name of the main JavaScript module that will be loaded.

- package.json: Defining the Node.js module

```
{  
  "author": "Brendan Dayley",  
  "name": "censorify",  
  "version": "0.1.1",  
  "description": "Censors words out of text",  
  "main": "censortext",  
  "dependencies": {},  
  
}
```

- Create a file named **README.md** in the **.../censorify** folder. You can put whatever read me instructions you want in this file.
- Navigate to the **.../censorify** folder in a console window and **run the npm pack** command to build a local package module.
- The **npm pack** command creates a **censorify-0.1.1.tgz** file in the **.../censorify** folder. This is your first Node.js Packaged Module

Publishing a Node.js Packaged Module to the NPM Registry

- When modules are published to the NPM registry, they are accessible to everyone using the NPM manager utility.
- This allows you to distribute your modules and applications to others more easily.
- The following steps describe the process of publishing the module to the NPM registry.
- After creating module,
 - Create a public repository to contain the code for the module. Then push the contents of the .../censorify folder up to that location.
 - The following is an example of a Github repository URL:
<https://github.com/username/projectname/directoryName>

- Create an account at <https://npmjs.org/signup>.
- Use the `npm adduser` command from a console prompt to add the user you created to the environment.
- Type in the username, password, and email that you used to create the account in step 2.
- Modify the `package.json` file to include the new repository information and any keywords that you want made available in the registry.

package.json: Defining the Node.js module that includes the repository and keywords information

```
{
  "author": "Brad Dayley",
  "name": "censorify",
  "version": "0.1.1",
  "description": "Censors words out of text",
  "main": "censortext",
  "repository": {
    "type": "git",
    "url": "Enter your github url"  },
  "keywords": [ "censor", "words" ],
  "dependencies": {},
  "engines": {
    "node": "*"  }
}
```

- Publish the module using the following command from the .../censor folder in the console:

npm publish

- Once the package has been published you can search for it on the NPM registry and use the npm install command to install it into your environment.
- To remove a package from the registry make sure that you have added a user with rights to the module to the environment using npm adduser and then execute the following command

npm unpublish <project name>

- For example, the following command unpublishes the censorify module:

npm unpublish censorify

- In some instances you cannot unpublish the module without using the --force option. This option forces the removal and deletion of the module from the registry.
- For example:

npm unpublish censorify --force

- **Using a Node.js Packaged Module in a Node.js Application**
- This section provides an example of actually using a Node.js module inside your Node.js applications.
- Node.js makes this simple: All you need to do is install the NPM into your application structure and then use the `require()` method to load the module.
- The `require()` method accepts either an installed module name or a path to a .js file located on the file system. For example:

```
require("censorify")
```

```
require("./lib/utils.js")
```


- Create a project folder named .../readwords.
- From a console prompt inside the .../readwords folder, use the following command to install the censorify module from the censorify- 0.1.1.tgz package you created earlier:
npm install .../censorify/censorify-0.1.1.tgz
- Or if you have published the censorify module, you can use the standard command to download and install it from the NPM registry:
npm install censorify
- Verify that a folder named node_modules is created along with a subfolder named censorify.
- Create a file named .../readwords/readwords.js.
- Add the following contents to the readwords.js file.

Notice that a **require()** call loads the **sensorify module** and assigns it to the variable **sensor**.

Then the **sensor variable** can be used to invoke the **getCensoredWords()**, **addCensoredWords()**, and **sensor()** functions from the **sensorify module**.

```
var sensor = require("sensorify");  
console.log(sensor.getCensoredWords());  
console.log(sensor.sensor("Some very sad, bad and mad text."));  
sensor.addCensoredWord("gloomy");  
console.log(sensor.getCensoredWords());  
console.log(sensor.sensor("A very gloomy day."));
```

Run the readwords.js application using the node readwords.js command and view the output shown in the following code block.

Notice that the censored words are replaced with **** and that the new censored word gloomy is added to the censorify module instance censor.

[Click here to view code image](#)

```
C:\nodeCode\ch03\readwords>node readwords [ 'sad', 'bad', 'mad' ]
```

```
Some very *****, *****, and ***** text.
```

```
[ 'sad', 'bad', 'mad', 'gloomy' ]
```

```
A very *** day.
```

Using Events

Objective : To understand the events, call backs in node js.

Course Outcome : Students are able to explain the node js events and call backs

Understanding the Node.js Event Model

- Node.js applications are run in a **single-threaded event-driven model**.
- **No concept of multiple threads** in node js.

Comparing Event Callbacks and Threaded Models

- In the **traditional threaded web model**, a **request comes in** to the webserver and is **assigned to an available thread**.
- Then the **handling of work** for that request continues on that thread until the request is complete and a response is sent.

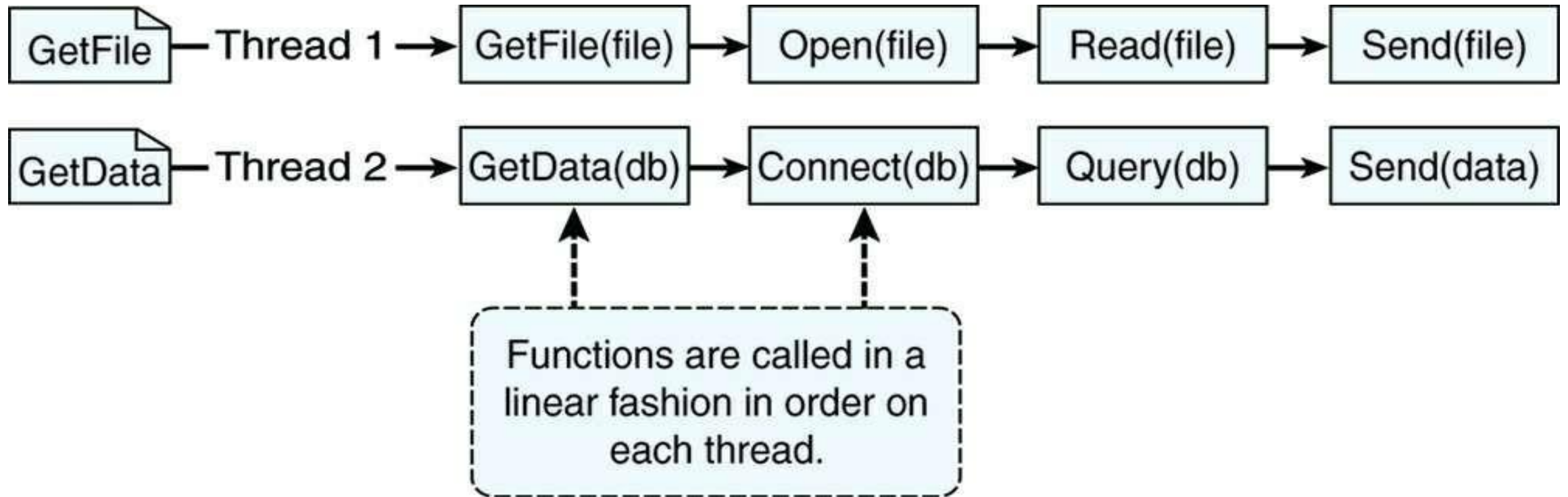


Figure 4.1 Processing two requests on individual threads using the threaded model

- [Figure 4.1](#) illustrates the threaded model processing two requests, GetFile and GetData.
- The GetFile request first opens the file, reads the contents, and then sends the data back in a response. All this occurs in order on the same thread.
- The GetData request connects to the DB, queries the necessary data, and then sends the data in the response.

The Node.js event model **does things differently.**

Instead of executing all the work for each request on individual threads, **work is added to an event queue and then picked up by a single thread** running an event loop.

The event loop **grabs the top item in the event queue**, executes it, and then grabs the next item.

When executing blocking I/O, instead of calling the function directly, the **function is added to the event queue along with a callback** that is executed after the function completes.

When all events on the Node.js event queue have been executed, the **Node application terminates.**

How Callbacks Work in Node.js

- Node.js callbacks are a special type of function passed as an argument to another function.
- They're called when the function that contains the callback as an argument completes its execution, allowing the code in the callback to run.

Syntax

```
function function_name(argument, callback)
```

Example

```
var fs = require('fs');
var rs = fs.createReadStream('./demofile.txt');
rs.on('open', function () {
  console.log('The file is open');
});
```

To include the built-in Events module use the `require()` method.

In addition, **all event properties and methods** are an instance of an **EventEmitter object**.

To be able to access these properties and methods, **create an EventEmitter object**:

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

- [Figure 4.2](#) illustrates the way Node.js handles the GetFile and GetData requests.
- The GetFile and GetData requests are added to the event queue.
- Node.js first picks up the GetFile request, executes it, and then completes it by adding the Open() callback function to the event queue.
- Next, it picks up the GetData request, executes it, and completes by adding the Connect() callback function to the event queue.
- This continues until there are no callback functions to be executed.
- Notice in [Figure 4.2](#) that the events for each thread do not necessarily follow a direct interleaved order.
- For example, the Connect request takes longer to complete than the Read request, so Send(file) is called before Query(db).

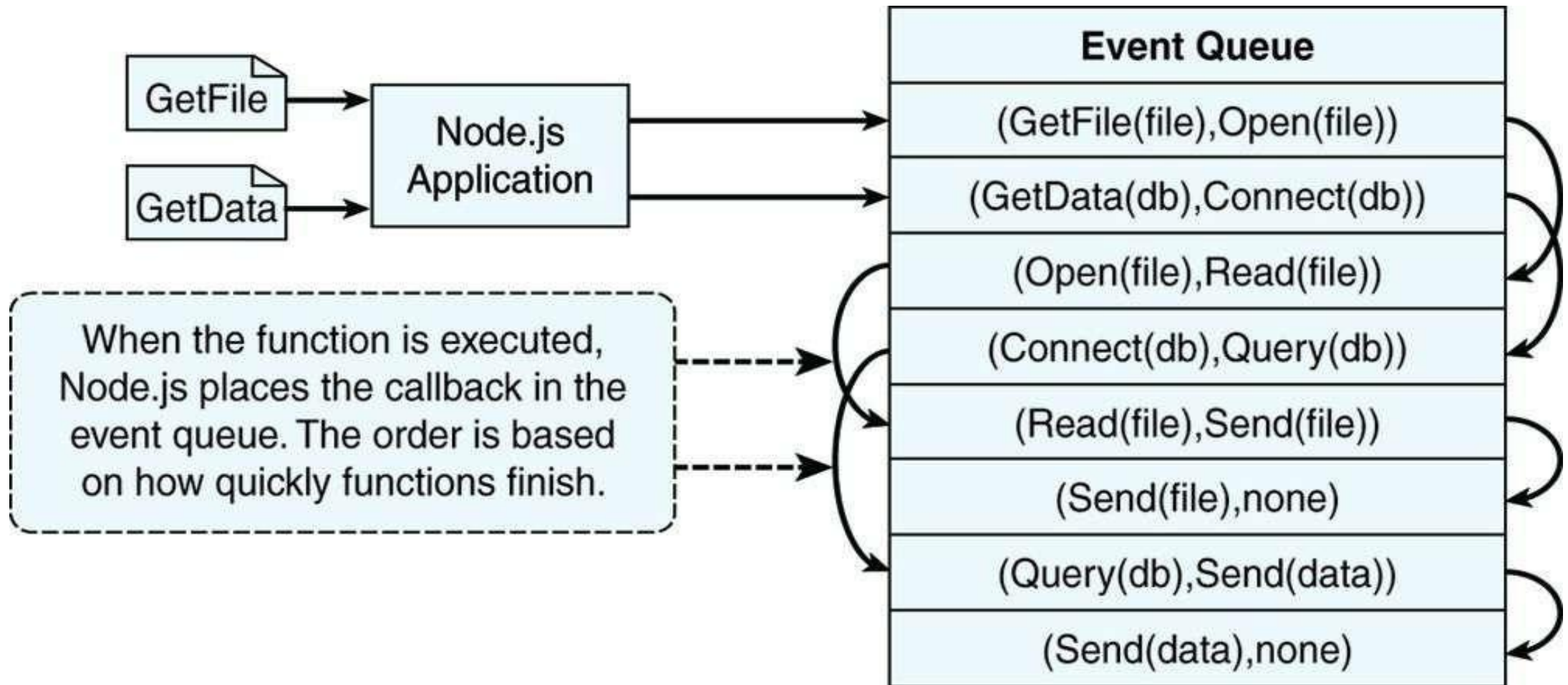


Figure: Processing two requests on a single event-driven thread using the Node.js event model

Blocking I/O in Node.js

Blocking I/O stops the execution of the current thread and waits for a response before continuing.

Some examples of blocking I/O are

- Reading a file
- Querying a database
- Socket request
- Accessing a remote service

The reason Node.js uses event callbacks is not to have to wait for blocking I/O.

Node.js implements a thread pool in the background.

When an event that requires blocking I/O is retrieved from the event queue, Node.js retrieves a thread from the thread pool and executes the function there instead of on the main event loop thread.

This prevents the blocking I/O from holding up the rest of the events in the event queue.

- The function executed on the blocking thread can still add events back to the event queue to be processed.
- For example, a database query call is typically passed a callback function that parses the results and may schedule additional work on the event queue before sending a response.
- [Figure 4.3](#) illustrates the full Node.js event model including the event queue, event loop, and thread pool.
- Notice that the event loop either executes the function on the event loop thread itself or, for blocking I/O, it executes the function on a separate thread.

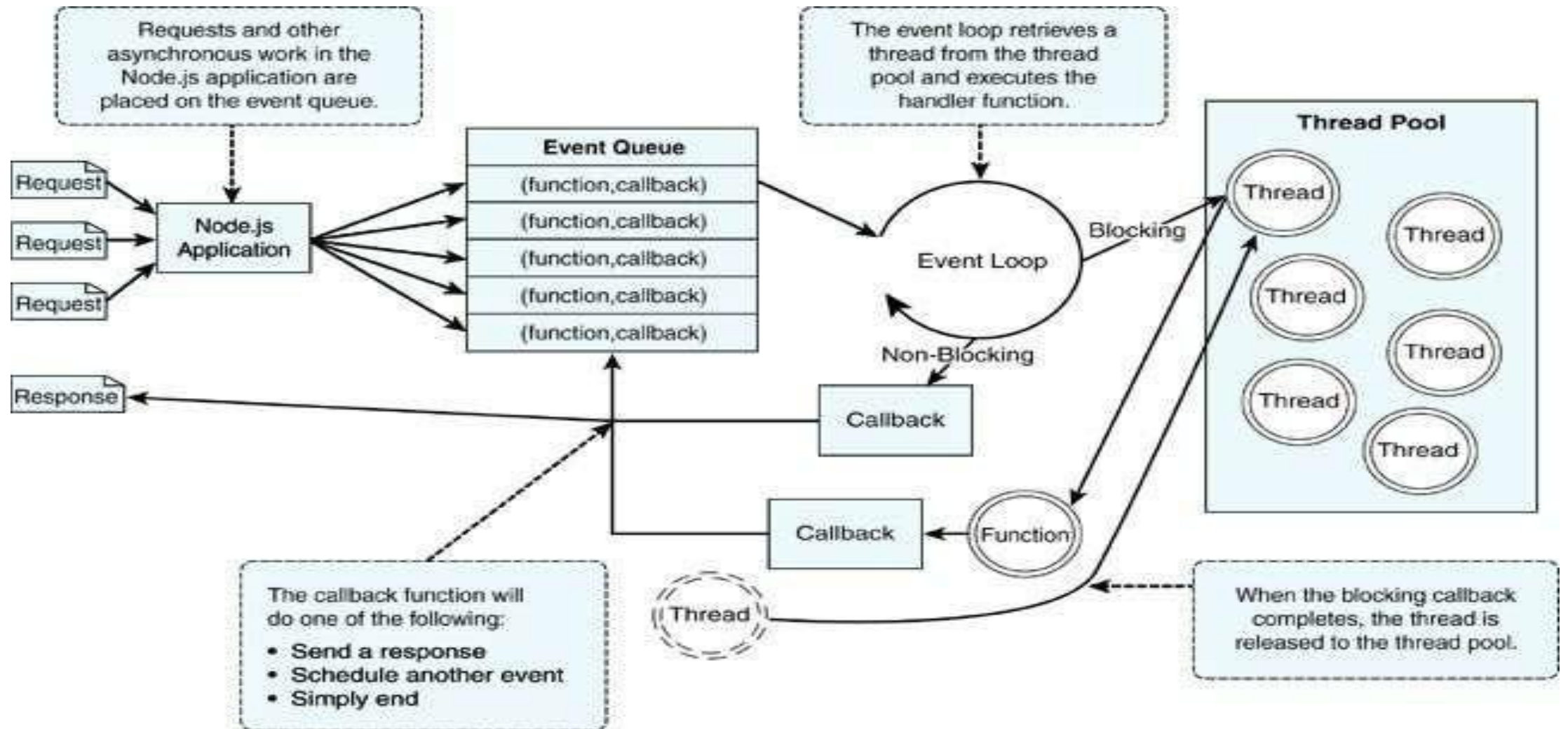


Figure 4.3 In the Node.js event model, work is added as a function with callback to the event queue, and then picked up on the event loop thread. The function is then executed on the event loop thread in the case of non-blocking, or on a separate thread in the case of blocking

- First, you are limited by the number of clones.
- What if you only have five clones? To talk with a sixth person, one clone must completely finish its conversation.
- The second problem is the limited number of CPUs (or “brains”) that the threads (“clones”) must share.
- This means that clones sharing the same brain have to stop talking/listening while other clones are using the brain.
- You can see that there really isn’t a benefit to having clones when they freeze while the other clones are using the brain.

- The Node.js event model acts more like real life when compared to the conversation example.
- First, Node.js applications run on a single thread, which means there is only one of you, no clones.
- Each time a person asks you a question, you respond as soon as you can.
- Your interactions are completely event driven, and you move naturally from one person to the next. Therefore, you can have as many conversations going on at the same time as you want by bouncing between individuals.
- Second, your brain is always focused on the person you are talking to since you aren't sharing it with clones.

How does Node.js handle blocking I/O requests?

- Think about when someone asks you a question that you have to think about.
- You can still interact with others at the party while trying to process that question in the back of your mind.
- That processing may impact how fast you interact with others, but you are still able to communicate with several people while processing the longer-lived thought.

Adding Work to the Event Queue

To leverage the scalability and performance of the event model break work up into chunks that can be performed as a series of callbacks.

Once you have designed your code correctly, you can then use the event model to schedule work on the event queue.

In Node.js applications, work is scheduled on the event queue by passing a callback function using one of these methods:

- Make a call to one of the blocking I/O library calls such as writing to a file or connecting to a database.
- Add a built-in event listener to a built-in event such as an `http.request` or `server.connection`.
- Create your own event emitters (to handle events) and add custom listeners to them.
- Use the `process.nextTick` option to schedule work to be picked up on the next cycle of the event loop.
- Use timers to schedule work to be done after a particular amount of time or at periodic intervals.

- The Node.js `process.nextTick` function interacts with the event loop in a special way
- When we pass a function to `process.nextTick()`, we instruct the engine to invoke this function at the end of the current operation, before the next event loop tick starts:

```
process.nextTick(() => {  
  // do something  
});
```

The event loop is busy processing the current function code. When this operation ends, the JS engine runs all the functions passed to `nextTick` calls during that operation.

```
console.log("Hello => number 1");
```

```
setImmediate(() => {  
  console.log("Running before the timeout => number 3");  
});
```

```
setTimeout(() => {  
  console.log("The timeout running last => number 4");  
}, 0);
```

```
process.nextTick(() => {  
  console.log("Running at next tick => number 2");  
});
```

Hello => number 1
Running at next tick => number 2
Running before the timeout =>
number 3
The timeout running last =>
number 4

Implementing Timers

- A useful feature of Node.js and JavaScript is the ability to delay the execution of code for a period of time.
 - This can be useful for cleanup or refresh work that you do not want to always be running.
 - There are three types of timers you can implement in Node.js: timeout, interval, and immediate.
-
- The following sections describe each of these timers and how to implement them in your code.

Delaying Work with **Timeouts**

- Timeout timers are used to delay work for a specific amount of time.
- When that time expires, the callback function is executed and the timer goes away.
- Timeout timers are created using the `setTimeout(callback, delayMilliseconds, [args])` method built into Node.js.
- When you call `setTimeout()`, the callback function is executed after delay `Milliseconds` expires. For example, the following executes `myFunc()` after 1 second:

```
setTimeout(myFunc, 1000);
```

- The `setTimeout()` function returns a timer object ID. You can pass this ID to `clearTimeout(timeoutId)` at any time before the `delayMilliseconds` expires to cancel the timeout function. For example:

```
myTimeout = setTimeout(myFunc, 100000);  
... clearTimeout(myTimeout);
```


- [Listing 4.1](#) implements a series of simple timeouts that call the `setTimeout()` function, which outputs the number of milliseconds since the timeout was scheduled. Notice that it doesn't matter which order `setTimeout()` is called; the results, shown in [Listing 4.1](#) Output, are in the order that the delay expires.

simple_timer.js: Implementing a series of timeouts at various intervals

```
function simpleTimeout(consoleTimer){  
  console.timeEnd(consoleTimer);  
}  
  
console.time("twoSecond");  
setTimeout(simpleTimeout, 2000, "twoSecond");  
console.time("oneSecond");  
setTimeout(simpleTimeout, 1000, "oneSecond");  
console.time("fiveSecond");  
setTimeout(simpleTimeout, 5000, "fiveSecond");  
console.time("50MilliSecond");  
setTimeout(simpleTimeout, 50, "50MilliSecond");
```

<Listing First>

Listing 4.1 Output simple_timer.js:

```
C:\books\node\ch04> node  
simple_timer.js
```

```
50MilliSecond: 50.489ms
```

```
oneSecond: 1000.688ms
```

```
twoSecond: 2000.665ms
```

```
fiveSecond: 5000.186ms
```

Performing Periodic Work with Intervals

- Interval timers are used to perform work on a regular delayed interval.
- When the delay time expires, the callback function is executed and is then rescheduled for the delay interval again.
- Use intervals for work that needs to be performed on a regular basis.
- Interval timers are created using the `setInterval(callback, delayMilliseconds, [args])` method built into Node.js.
- When you call `setInterval()`, the callback function is executed every interval after `delayMilliseconds` has expired. For example, the following executes `myFunc()` every second:

```
setInterval(myFunc, 1000);
```

- The `setInterval()` function returns a timer object ID. You can pass this ID to `clearInterval(intervalId)` at any time before the `delayMilliseconds` expires to cancel the timeout function. For example:

```
myInterval = setInterval(myFunc, 1000000);  
... clearInterval(myInterval);
```

- Listing 4.2 implements a series of simple interval callbacks that update the values of the variables x, y, and z at different intervals.
- Notice that the values of x, y, and z are changed differently because the interval amounts are different, with x incrementing twice as fast as y, which increments twice as fast as z, as shown in Listing 4.2 Output.

Listing 4.2 simple_interval.js: Implementing a series of update callbacks at various intervals

```
var x=0, y=0, z=0;
function displayValues(){
  console.log("X=%d; Y=%d; Z=%d", x, y, z); }
function updateX(){
  x += 1;
}
function updateY(){
  y += 1;
}
function updateZ(){
  z += 1;
  displayValues(); }
setInterval(updateX, 500);
setInterval(updateY, 1000);
setInterval(updateZ, 2000);
```

simple_interval.js: Interval
functions executed at different
delay amounts

```
C:\books\node\ch04> node
simple_interval.js
x=3; y=1; z=1
x=7; y=3; z=2
x=11; y=5; z=3
x=15; y=7; z=4
x=19; y=9; z=5
x=23; y=11; z=6
```

Performing Immediate Work with an Immediate Timer

- Immediate timers are used to **perform work on a function as soon as the I/O event callbacks are executed**, but before any timeout or interval events are executed.
- Use immediate timers **to yield long-running execution segments** to other callbacks **to prevent starving the I/O events**.
- Immediate timers are created using the **setImmediate(callback,[args])** method built into Node.js.
- When you call setImmediate(), the callback function is placed on the event queue and popped off once for each iteration through the event queue loop after I/O events have a chance to be called.
- For example, the following schedules myFunc() to execute on the next cycle through the event queue:

```
setImmediate(myFunc());
```

- The setImmediate() function returns a timer object ID. You can pass this ID to clearImmediate(immediateId) at any time before it is picked up off the event queue. For example:

```
myImmediate = setImmediate(myFunc);
```

```
...
```

```
clearImmediate(myImmediate);
```

Dereferencing Timers from the Event Loop

- Often **you do not want timer event callbacks to continue** to be scheduled when they are the only events left in the event queue.
- The **unref()** function available in the object returned by `setInterval` and `setTimeout` allows you to notify the **event loop to not continue** when these are the only events on the queue.

For example, the following dereferences the `myInterval` interval timer:

```
myInterval = setInterval(myFunc);  
myInterval.unref();
```

- If for some reason you later **do not want the program to terminate** if the interval function is the only event left on the queue, you can use the `ref()` function to re-reference it:

```
myInterval.ref();
```

Using nextTick to Schedule Work

- A useful method of scheduling work on the event queue is the `process.nextTick(callback)` function.
- This function schedules work to be run on the next cycle of the event loop. Unlike the `setImmediate()` method, `nextTick()` executes before the I/O events are fired.
- This can result in starvation of the I/O events, so Node.js limits the number of `nextTick()` events that can be executed each cycle through the event queue by the value of `process.maxTickDepth`, which defaults to 1000.


```
var fs = require("fs");
fs.stat("nexttick.js", function(){
  console.log("nexttick.js Exists"); });
setImmediate(function(){
  console.log("Immediate Timer 1 Executed"); });
setImmediate(function(){
  console.log("Immediate Timer 2 Executed"); });
process.nextTick(function(){
  console.log("Next Tick 1 Executed"); });
process.nextTick(function(){
  console.log("Next Tick 2 Executed"); });
```

Listing 4.3 Output nexttick.js: Executing the nextTick() calls first

- `c:\books\node\ch04>node nexttick.js`
- Next Tick 1 Executed
- Next Tick 2 Executed
- Immediate Timer 1 Executed
- Immediate Timer 2 Executed
- nexttick.js Exists

Implementing Event Emitters and Listeners

- creating your own custom events as well as implementing listener callbacks that get implemented when an event is emitted.

Adding Custom Events to Your JavaScript Objects

- Events are emitted using an EventEmitter object.
- This object is included in the events module.
- The `emit(eventName, [args])` function triggers the eventName event and includes any arguments provided.
- The following code snippet shows how to implement a simple event emitter:

```
var events = require('events');  
var emitter = new events.EventEmitter();  
emitter.emit("simpleEvent");
```

- Two most common **ways to create an event emitter** in Node.js.
- The first is to use an **event emitter object directly**, and the second is to create an object that extends the event emitter object.

- To add events directly to your JavaScript objects, you need to inherit the EventEmitter functionality in your object by calling events.
- `EventEmitter.call(this)` in your object instantiation as well as adding the `events.EventEmitter.prototype` to your object's prototyping.
- For example:

```
Function MyObj()  
{  
  Events.EventEmitter.call(this);  
}  
MyObj.prototype.__proto__ = events.EventEmitter.prototype;
```

Can emit events directly from instances of your object. For example:

```
var myObj = new MyObj();  
myObj.emit("someEvent");
```

Adding Event Listeners to Objects

Once you have an instance of an object that can emit events, you can add listeners for the events that you care about.

Listeners are added to an EventEmitter object using one of the following functions:

.addListener(eventName, callback): Attaches the callback function to the object's listeners.

.on(eventName, callback): Same as .addListener().

.once(eventName, callback): Only the first time the eventName event is triggered, the callback function is placed in the event queue to be executed.

For example, to add a listener to an instance of the MyObject EventEmitter class defined in the previous section you would use the following:

```
function myCallback(){  
  ...  
}  
var myObject = new MyObj();  
myObject.on("someEvent", myCallback);
```

Removing Listeners from Objects

Listeners are useful and vital parts of Node.js programming.

However, they do cause overhead, and you should **use them only when necessary**.

Node.js **provides server helper functions on the EventEmitter object** that allow you to manage the listeners that are included.

These include

.listeners(eventName): Returns an array of listener functions attached to the eventName event.

.setMaxListeners(n): Triggers a warning if more than n listeners are added to an EventEmitter object. The default is 10.

.removeListener(eventName, callback): Removes the callback function from the eventName event of the EventEmitter object.

Implementing Event Listeners and Event Emitters

- [Listing 4.4](#) demonstrates the process of implementing listeners and custom event emitters in Node.js.
- The Account object is extended to inherit from the EventEmitter class and provides two methods to deposit and withdraw that both emit the balanceChanged event.
- Then in lines 15–31, three callback functions are implemented that are attached to the Account object instance balanceChanged event and display various forms of data.
- Notice that the checkGoal(acc, goal) callback is implemented a bit differently than the others.
- This was done to illustrate how to pass variables into an event listener function when the event is triggered. The results of executing the code are shown in [Listing 4.4](#) Output


```

var events = require('events');
function Account()
{
this.balance = 0;
events.EventEmitter.call(this);
this.deposit = function(amount){
this.balance += amount;
this.emit('balanceChanged');
};
this.withdraw = function(amount){
this.balance -= amount;
this.emit('balanceChanged');
}; }
Account.prototype. proto  =
events.EventEmitter.prototype;
function displayBalance(){
console.log("Account balance: $%d",
this.balance); }

```

```

function checkOverdraw(){
if (this.balance < 0){
console.log("Account overdrawn!!!");    } }
function checkGoal(acc, goal){
if (acc.balance > goal){
console.log("Goal Achieved!!!");    } }
var account = new Account();
account.on("balanceChanged",
displayBalance);
account.on("balanceChanged",
checkOverdraw);
account.on("balanceChanged", function(){
    checkGoal(this, 1000); });
account.deposit(220);
account.deposit(320);
account.deposit(600);
account.withdraw(1200);

```

Listing 4.4 Output emitter_listener.js: The account statements output by the listener callback functions

```
C:\books\node\ch04>node emitter_listener.js
```

```
Account balance: $220
```

```
Account balance: $540
```

```
Account balance: $1140
```

```
Goal Achieved!!!
```

```
Account balance: $-60
```

```
Account overdrawn!!!
```

Passing Additional Parameters to Callbacks

Most callbacks have automatic parameters passed to them, such as an error or result buffer.

[Listing 4.5](#) illustrates implementing callback parameters.

There are two sawCar event handlers.

Note that the sawCar event only emits the make parameter.

Notice that the emitter.emit() function also can accept additional parameters; in this case, make is added as shown in line 5.

The first event handler on line 16 implements the logCar(make) callback handler.

To add a color for logColorCar(), an anonymous function is used in the event handler defined in lines 17–21.

A randomly selected color is passed to the call logColorCar(make, color).

You can see the output in [Listing 4.5](#) Output.

callback_parameter.js: Creating an anonymous function to add additional parameters not emitted by the event

```
var events = require('events');
function CarShow()
{
  events.EventEmitter.call(this);
  this.seeCar = function(make){
    this.emit('sawCar', make);
  }; }
CarShow.prototype. proto  =
events.EventEmitter.prototype;
```

```
var show = new CarShow();
function logCar(make){
  console.log("Saw a " + make); }
```

```
function logColorCar(make, color){
  console.log("Saw a %s %s", color, make); }
```

```
show.on("sawCar", logCar);
show.on("sawCar", function(make)
{
  var colors = ['red', 'blue', 'black'];
  var color =
    colors[Math.floor(Math.random()*3)];
  logColorCar(make, color); }));
```

```
show.seeCar("Ferrari");
show.seeCar("Porsche");
show.seeCar("Bugatti");
show.seeCar("Lamborghini");
show.seeCar("Aston Martin");
```

Output callback_parameter.js:

The results of adding a color parameter to the callback

- C:\books\node\ch04>node callback_parameter.js

Saw a Ferrari

Saw a blue Ferrari

Saw a Porsche

Saw a black Porsche

Saw a Bugatti

Saw a red Bugatti

Saw a Lamborghini

Saw a black Lamborghini

Saw a Aston Martin

Saw a black Aston Martin

- **Implementing Closure in Callbacks**

- An interesting problem that asynchronous callbacks have is that of closure.
- Closure is a JavaScript term that indicates that variables are bound to a function's scope and not the parent function's scope.
- When you execute an asynchronous callback, the parent function's scope may have changed; for example, when iterating through a list and altering values in each iteration.
- If your callback needs access to variables in the parent function's scope, then you need to provide closure so that those values are available when the callback is pulled off the event queue.
- A basic way of doing that is by encapsulating the asynchronous call inside a function block and passing in the variables that are needed.

```
function logCar(logMsg, callback)
{
    process.nextTick(function() {
        callback(logMsg);});
}
```

```
var cars = ["Ferrari", "Porsche", "Bugatti"];
```

```
for (var idx in cars)
{
    var message = "Saw a " + cars[idx];
    logCar(message, function(){
        console.log("Normal Callback: " +
message);    });
}
```

```
for (var idx in cars)
{
    var message = "Saw a " + cars[idx];
    logCar(msg, function(){
        console.log("Closure Callback: " + msg);    });
}
```

Output

```
C:\books\node\ch04>node callback_closure.js
Normal Callback: Saw a Bugatti
Normal Callback: Saw a Bugatti
Normal Callback: Saw a Bugatti
Closure Callback: Saw a Ferrari
Closure Callback: Saw a Porsche
Closure Callback: Saw a Bugatti
```



```
var events = require('events');  
var EventEmitter = events.EventEmitter;  
var chat = new EventEmitter();  
var users = [], chatlog = [];  
chat.on('message', function(message)  
{  
  chatlog.push(message);  
});
```

```
chat.on('join', function(nickname)  
{  
  users.push(nickname);  
});  
// Emit events here  
chat.emit('join', "Kate");  
chat.emit('message', "Hi, I'm Kate.");
```

Listing 4.6 Output `callback_closure.js`: Adding a closure wrapper function allows the asynchronous callback to access necessary variables

Chaining Callbacks

- With asynchronous functions you are not guaranteed the order that they will run if two are placed on the event queue.
- The best way to resolve that is to implement callback chaining by having the callback from the asynchronous function call the function again until there is no more work to do.
- That way the asynchronous function is never on the event queue more than once.

A list of items is passed into the function `logCars()`, the asynchronous function `logCar()` is called, and then the `logCars()` function is used as the callback when `logCar()` completes.

Thus only one version of `logCar()` is on the event queue at the same time.

Listing 4.7 `callback_chain.js`: Implementing a callback chain where the callback from an anonymous function calls back into the initial function to iterate through a list

```
function logCar(car, callback)
{
  console.log("Saw a %s", car);
  if(cars.length){
    process.nextTick(function(){
      callback();
    });
  }
}

function logCars(cars)
{
  var car = cars.pop();
  logCar(car, function(){
    logCars(cars);  });
}

var cars = ["Ferrari", "Porsche", "Bugatti",
  "Lamborghini", "Aston Martin"];
logCars(cars);
```

Listing 4.7 Output callback_chain.js: Using an asynchronous callback chain to iterate through a list

```
C:\books\node\ch04>node callback_chain.js
```

```
Saw a Aston Martin
Saw a Lamborghini
Saw a Bugatti
Saw a Porsche
Saw a Ferrari
```

```
function first(data, cb1, cb2, cb3)
{
  console.log("first()", data);
  cb1(data,cb2, cb3);
}

function second(data, cb1, cb2)
{
  console.log("second()",data);
  cb1(data, cb2);
}
```

```
function third(data, cb)
{
  console.log("third()",data);
  cb(data);
}

function last(data)
{
  console.log("last() ", data);
}

first("THEDATA", second, third, last);
first("THEDATA2", second, last);
first("THEDATA3", second);
```

Handling Data I/O in Node.js

- For all the file system calls loaded the fs module, for example:

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

- **Synchronous Versus Asynchronous File System Calls**

- The fs module provided in Node.js makes almost all functionality available in two forms:

asynchronous and synchronous.

For example, there is the asynchronous form `write()` and the synchronous form `writeSync()`.

- Synchronous file system calls block until the call completes and then control is released back to the thread.
- This has advantages but can also cause severe performance issues in Node.js if synchronous calls block the main event thread.
- Therefore, synchronous file system calls should be limited in use when possible.
- Asynchronous calls are placed on the event queue to be run later.
- This allows the calls to fit into the Node.js event model; however, this can be tricky when executing your code because the calling thread continues to run before the asynchronous call gets picked up by the event loop.

Opening and Closing Files

- Node provides synchronous and asynchronous methods for opening files.
- Once a file is opened, you can read data from it or write data to it depending on the flags used to open the file.
- To open files in a Node.js app, use one of the following statements for asynchronous or synchronous:

`fs.open(path, flags, [mode], callback)`

`fs.openSync(path, flags, [mode])`

Working with JSON

- One of the most common data types that you work with when implementing Node.js web applications and services is JSON (JavaScript Object Notation).
- JSON is a lightweight method to convert JavaScript objects into a string form and then back again.

There are **several reasons to use JSON** to serialize your JavaScript objects over XML including the following:

- JSON is much more efficient.
- Serializing/deserializing JSON is faster than XML.
- JSON is easier to read from a developer's perspective because it is similar to JavaScript syntax.

Converting JSON to JavaScript Objects

- A JSON string represents the JavaScript object in string form.
- The string syntax is similar to code, making it easy to understand.
- Use `JSON.parse(string) method` to convert a string that is properly formatted with JSON into a JavaScript object.

```
var accountStr = '{"name":"Jedi", "members":["Yoda","Obi Wan"],  
"number":34512, "location": "A galaxy far, far away"}';  
var accountObj = JSON.parse(accountStr);  
console.log(accountObj.name);  
console.log(accountObj.members);
```

The preceding code outputs the following:

Jedi

['Yoda', 'Obi Wan']

Converting JavaScript Objects to JSON

- Node also allows you to convert a JavaScript object into a properly formatted JSON string.
- Thus the **string form can be stored in a file or database**, sent across an HTTP connection, or written to a stream/buffer.
- Use the **JSON.stringify(text) method to parse JSON text** and generate a JavaScript object

Output

```
var accountObj = {  
  name: "Baggins",  
  number: 10645,  
  members: ["Frodo, Bilbo"],  
  location: "Shire"  
};  
var accountStr = JSON.stringify(accountObj);  
console.log(accountStr);
```

The preceding code outputs the following:

```
{"name":"Baggins","number":10645,"members":["Frodo,  
Bilbo"],"location":"Shire"}
```

Using the Buffer Module to Buffer Data

- While JavaScript is Unicode friendly, it is not good at managing binary data.
- However, binary data is useful when implementing some web applications and services.
- For example:
 - Transferring compressed files
 - Generating dynamic images
 - Sending serialized binary data

- Buffered data
- Create buffers
- Write to buffer
- Read from buffer
- Determining buffer length
- Copying buffer
- Slicing buffer
- Concatenating buffer

Understanding Buffered Data

- Buffered data is made up of a series of octets in big endian or little endian format.
- Node.js provides the Buffer module that gives you the functionality to create, read, write, and manipulate binary data in a buffer structure.
- The Buffer module is global, so you do not need to use the require() statement to access it.
- Buffered data is stored in a structure similar to that of an array. Therefore a Buffer cannot be resized.
- When converting buffers to and from strings, you need to specify the explicit encoding method to be used.

Table 5.1 Methods of encoding between strings and binary buffers

Method	Description
<code>utf8</code>	Multi-byte encoded Unicode characters used in documents and webpages.
<code>utf16le</code>	Little endian encoded Unicode characters of 2 or 4 bytes.
<code>ucs2</code>	Same as <code>utf16le</code> .
<code>base64</code>	Base64 string encoding.
<code>Hex</code>	Encode each byte as two hexadecimal characters.

Big Endian and Little Endian

- Binary data in buffers is stored as a series of octets or a sequence of eight 0s and 1s.
 - It can be read as a single byte or as a word containing multiple bytes.
 - Endian defines the ordering of significant bits when defining the word.
 - Big endian stores the least significant word first, and little endian stores the least significant word last.
-
- For example,
 - the words 0x0A 0x0B 0x0C 0x0D would be stored in the buffer as [0x0A, 0x0B, 0x0C, 0x0D] in big-endian
but as [0x0D, 0x0C, 0x0B, 0x0A] in little endian.

Creating Buffers

- Buffer objects are actually raw memory allocations; therefore, their size must be determined when they are created.
- The three methods for creating Buffer objects using the new keyword are
 - **new Buffer(sizeInBytes)**
 - **new Buffer(octetArray)**
 - **new Buffer(string, [encoding])**
- For example, the following lines of code define buffers using a byte size, octet buffer, and a UTF8 string:

```
var buf256 = new Buffer(256);
```

```
var bufOctets = new Buffer([0x6f, 0x63, 0x74, 0x65, 0x74, 0x73]);
```

```
var bufUTF8 = new Buffer("Some UTF8 Text \u00b6 \u30c6 \u20ac",  
'utf8');
```

Writing to Buffers

- The size of a Buffer object cannot be extended after it has been created, but can write data to any location in the buffer.
- Table describes the three methods you can use when writing to buffers.

Table 5.2 Methods of writing from `Buffer` objects

Method	Description
<code>buffer.write(string, [offset], [length], [encoding])</code>	Writes <code>length</code> number of bytes from the <code>string</code> starting at the <code>offset</code> index inside the buffer using encoding.
<code>buffer[offset] = value</code>	Replaces the data at index <code>offset</code> with the value specified.
<code>buffer.fill(value, [offset], [end])</code>	Writes the value to every byte in the buffer starting at the <code>offset</code> index and ending with the <code>end</code> index.
<code>writeInt8(value, offset, [noAssert])</code>	There is a wide range of methods for <code>Buffer</code> objects to write integers, unsigned integers, doubles, and floats of various sizes using little endian or big endian. <code>value</code> specifies the value to write, <code>offset</code> specifies the index to write to, and <code>noAssert</code> specifies whether to skip validation of the value and <code>offset</code> . <code>noAssert</code> should be left at the default <code>false</code> unless you are absolutely certain of correctness.
<code>writeInt16LE(value, offset, [noAssert])</code>	
<code>writeInt16BE(value, offset, [noAssert])</code>	
...	

buffer_write.js: Various ways to write to a Buffer object

```
buf = new Buffer(256);  
buf.fill(0);  
buf.write("add some text");  
console.log(buf.toString());  
buf.write("more text", 9, 9);  
console.log(buf.toString());  
buf[18] = 43;  
console.log(buf.toString());
```

- **Output buffer_write.js: Writing data from a Buffer object**

```
C:\books\node\ch05>node buffer_write.js  
add some text  
add some more text  
add some more text+
```

Reading from Buffers

- There are several methods for reading from buffers.
- The simplest is to use the `toString()` method to convert all or part of a buffer to a string.
- However, you can also access specific indexes in the buffer directly or by using `read()`.
- Also Node.js provides a `StringDecoder` object that has a `write(buffer)` method that decodes and writes buffered data using the specified encoding.

Table 5.3 Methods of reading from `Buffer` objects

Method	Description
<code>buffer.toString([encoding], [start], [end])</code>	Returns a string containing the decoded characters specified by encoding from the start index to the end index of the buffer. If start or end is not specified, then <code>toString()</code> uses the beginning or end of the buffer.
<code>stringDecoder.write(buffer)</code>	Returns a decoded string version of the buffer.
<code>buffer[offset]</code>	Returns the octet value in the buffer at the specified offset.
<code>readInt8(offset, [noAssert])</code> <code>readInt16LE(offset, [noAssert])</code> <code>readInt16BE(offset, [noAssert])</code> ...	There is a wide range of methods for <code>Buffer</code> objects to read integers, unsigned integers, doubles, and floats of various sizes using little endian or big endian. These functions accept the offset to read from an optional <code>noAssert</code> Boolean value that specifies whether to skip validation of the offset. <code>noAssert</code> should be left at the default <code>false</code> unless you are absolutely certain of correctness.

Listing 5.2 buffer_read.js: Various ways to read from a Buffer object

```
buf = new Buffer("Some UTF8 Text \u00b6 \u30c6 \u20ac", 'utf8');  
console.log(buf.toString());  
console.log(buf.toString('utf8', 5, 9));  
var StringDecoder = require('string_decoder').StringDecoder;  
var decoder = new StringDecoder('utf8');  
console.log(decoder.write(bufUTF8));  
console.log(buf[18].toString(16));  
console.log(buf.readUInt32BE(18).toString(16));
```

Listing 5.2 Output buffer_read.js: Reading data from a Buffer object

```
C:\books\node\ch05>node buffer_read.js  
Some UTF8 Text ¶ ¯ €  
UTF8  
Some UTF8 Text ¶ ¯ €  
e3  
e3838620
```

Determining Buffer Length

- A common task when dealing with buffers is **determining the length**, especially when you create a buffer dynamically from a string.
- The **length of a buffer can be determined by calling .length** on the Buffer object.
- To determine the byte length that a string takes up in a buffer you cannot use the **.length** property.
- Instead you need to use **Buffer.byteLength(string, [encoding])**.
- Note that there is a difference between the string length and byte length of a buffer.

To illustrate this consider the followings statements:

```
Buffer.byteLength("UTF8 text \u00b6", 'utf8');
```

//evaluates to 12

```
Buffer("UTF8 text \u00b6").length;
```

//evaluates to 12

Copying Buffers

- An important part of working with buffers is the ability to copy data from one buffer into another buffer.
- Node.js provides the `copy(targetBuffer, [targetStart], [sourceStart], [sourceIndex])` method on Buffer objects.
- The targetBuffer parameter is another Buffer object, and targetStart, sourceStart, and sourceEnd are indexes inside the source and target buffers.

```

var dept= new Buffer('Information Technology');
console.log(dept.toString());
// copy full buffer
var blank = new Buffer(26);
blank.fill();
console.log("Blank: " + blank.toString());
dept.copy(blank);
console.log("Blank: " + blank.toString());
// copy part of buffer
var dashes = new Buffer(26);
dashes.fill('-');
console.log("Dashes: " + dashes.toString());
dept.copy(dashes, 10, 10, 15);
console.log("Dashes: " + dashes.toString());

```

Information Technology

Blank:

Blank: Information Technology....

Dashes: -----

Dashes: -----n Tec-----

Slicing Buffer

- A *slice* is a section of a buffer between a starting index and an ending index.
- Slicing a buffer allows you to manipulate a specific chunk.
- Slices are created using the **slice([start], [end])** method, which returns a Buffer object that points to start index of the original buffer and has a length of end – start.
- A slice is different from a copy. If you edit a copy, the original does not change. If you edit a slice, the original does change.

buffer_slice.js: Creating and manipulating slices of a Buffer object

```
var numbers = new Buffer("123456789");  
console.log(numbers.toString());  
var slice = numbers.slice(3, 6);  
console.log(slice.toString());  
slice[0] = '#'.charCodeAt(0);  
slice[slice.length-1] = '#'.charCodeAt(0);  
console.log(slice.toString());  
console.log(numbers.toString());
```

Listing 5.4 Output

buffer_slice.js: Slicing and modifying a Buffer object

```
C:\books\node\ch05>node  
buffer_slice.js
```

```
123456789
```

```
456
```

```
#5#
```

```
123#5#789
```


Concatenating Buffers

- You can also concatenate two or more Buffer objects together to form a new buffer.
- The `concat(list, [totalLength])` method accepts an array of Buffer objects as the first parameter, and `totalLength` defines the maximum bytes in the buffer as an optional second argument.
- The Buffer objects are concatenated in the order they appear in the list, and a new Buffer object is returned containing the contents of the original buffers up to `totalLength` bytes.

buffer_concat.js: Concatenating Buffer objects

```
var af = new Buffer("African Swallow?");  
var eu = new Buffer("European Swallow?");  
var question = new Buffer("Air Speed Velocity of an ");  
console.log(Buffer.concat([question, af]).toString());  
console.log(Buffer.concat([question, eu]).toString());
```

Output buffer_concat.js: Concatenating Buffer objects

```
C:\books\node\ch05>node buffer_concat.js  
Air Speed Velocity of an African Swallow?  
Air Speed Velocity of an European Swallow?
```

Using the Stream Module to Stream Data

- Data streams are memory structures that are readable, writable, or both.
- The purpose of streams is to provide a common mechanism to transfer data from one location to another.
- They also expose events, such as when data is available to be read, when an error occurs, and so on.
- You can then register listeners to handle the data when it becomes available in a stream or is ready to be written to.

Readable Streams

Readable streams provide a mechanism to easily read data coming into the application from another source.

Some common examples of readable streams are HTTP responses on the client

- HTTP requests on the server
- fs read streams
- zlib streams
- crypto streams
- TCP sockets
- Child processes stdout and stderr
- process.stdin

Readable streams provide the `read([size])` method to read data where `size` specifies the number of bytes to read from the stream.

`read()` can return a `String`, `Buffer` or `null`.

Readable streams also expose the following events:

- **readable**: Emitted when a chunk of data can be read from the stream.
- **data**: Similar to `readable` except that when data event handlers are attached, the stream is turned into flowing mode, and the data handler is called continuously until all data has been drained.
- **end**: Emitted by the stream when data will no longer be provided.
- **close**: Emitted when the underlying resource, such as a file, has been closed.
- **error**: Emitted when an error occurs receiving data.

Table 5.4 Methods available on `Readable` stream objects

Method	Description
<code>read([size])</code>	Reads data from the stream. The data can be a <code>String</code> , <code>Buffer</code> , or <code>null</code> , meaning there is no more data left. If a <code>size</code> argument is read, then the data is limited to that number of bytes.
<code>setEncoding(encoding)</code>	Sets the encoding to use when returning <code>String</code> in the <code>read()</code> request.
<code>pause()</code>	This pauses data events from being emitted by the object.
<code>resume()</code>	The resumes data events being emitted by the object.
<code>pipe(destination, [options])</code>	This pipes the output of this stream into a <code>Writable</code> stream object specified by <code>destination</code> . <code>options</code> in a JavaScript object. For example, <code>{end:true}</code> ends the <code>Writable</code> destination when the <code>Readable</code> ends.
<code>unpipe([destination])</code>	Disconnects this object from the <code>Writable</code> destination.

- To implement your own custom Readable stream object, you need to first inherit the functionality for Readable streams.
- The simplest way to do that is to use the util module's inherits() method:

```
var util = require('util');  
util.inherits(MyReadableStream, stream.Readable);
```

Then you create an instance of the object call:

```
stream.Readable.call(this, opt);
```

- You also need to implement a _read() method that calls push() to output the data from the Readable object.
- The push() call should push either a String, Buffer, or null.

Writing Files

- The fs module provides four different ways to write data to files.
- Can write data to a file in a single call, write chunks using synchronous writes, write chunks using asynchronous writes, or stream writes through a Writable stream.
- Each of these methods accepts either a String or a Buffer object as input.

Simple File Write

- The simplest method for writing data to a file is to use one of the writeFile() methods. These methods write the full contents of a String or Buffer to a file.
- The following shows the syntax for the writeFile() methods:

fs.writeFile(path, data, [options], callback)

fs.writeFileSync(path, data, [options])

- **Writable Streams**

- Writable streams are designed to provide a mechanism to write data into a form
- that can easily be consumed in another area of code. Some common examples of
- Writable streams are
 - HTTP requests on the client
 - HTTP responses on the server
 - fs write streams
 - zlib streams
 - crypto streams
 - TCP sockets
 - Child process stdin
 - process.stdout, process.stderr

- Writable streams provide the `write(chunk, [encoding], [callback])` method to write data into the stream
- The `write()` function returns `true` if the data was written successfully. Writable streams also expose the following events:
 - **drain:** After a `write()` call returns `false`, the `drain` event is emitted to notify listeners when it is okay to begin writing more data.
 - **finish:** Emitted when `end()` is called on the Writable object; all data is flushed and no more data will be accepted.
 - **pipe:** Emitted when the `pipe()` method is called on a Readable stream to add this Writable as a destination.
 - **unpipe:** Emitted when the `unpipe()` method is called on a Readable stream to remove this Writable as a destination.

Table 5.5 Methods available on Writable stream objects

Method	Description
<code>write(chunk, [encoding], [callback])</code>	Writes the data chunk to the stream object's data location. The data can be a <code>String</code> or <code>Buffer</code> . If <code>encoding</code> is specified, then it is used to encode string data. If <code>callback</code> is specified, then it is called after the data has been flushed.
<code>end([chunk], [encoding], [callback])</code>	Same as <code>write()</code> , except it puts the <code>Writable</code> into a state where it no longer accepts data and sends the <code>finish</code> event.

To implement your own custom `Writable` stream object, you need to first inherit the functionality for `Writable` streams. The simplest way to do that is to use the `util` module's `inherits()` method:

```
var util = require('util');  
util.inherits(MyWritableStream, stream.Writable);
```

Then you create an instance of the object call:

```
stream.Writable.call(this, opt);
```

```
var fs = require("fs");
var data = '';
// Create a readable stream
var readerStream = fs.createReadStream('input.txt');
// Set the encoding to be utf8.
readerStream.setEncoding('UTF8');
// Handle stream events --> data, end
readerStream.on('data', function(chunk) {
    data += chunk;
});
readerStream.on('end',function() {
    console.log(data);
});
console.log("Program Ended");
```

```
var fs = require("fs");
var data = 'Simply Easy Learning';
// Create a writable stream
var writerStream = fs.createWriteStream('output.txt');
// Write the data to stream with encoding to be utf8
writerStream.write(data,'UTF8');
// Mark the end of file
writerStream.end();
// Handle stream events --> finish
writerStream.on('finish', function() {
    console.log("Write completed.");
});
console.log("Program Ended");
```

Implementing HTTP Services in Node.js

Processing URLs

- The Uniform Resource Locator (URL) acts as an address label for the HTTP server to handle requests from the client.
- It provides all the information needed to get the request to the correct server on a specific port and access the proper data.
- The URL can be broken down into several different components, each providing a basic piece of information for the webserver on how to route and handle the HTTP request from the client.

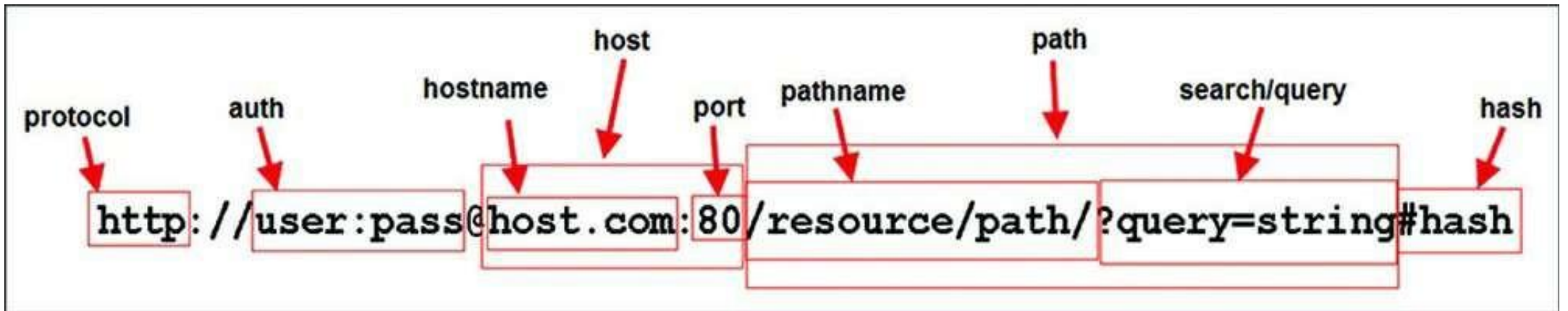


Figure : Basic components that can be included in a URL

- **Understanding the URL Object**

- To create a URL object from the URL string, pass the URL string as the first parameter to the following method:

`url.parse(urlStr, [parseQueryString], [slashesDenoteHost])`

- The `parseQueryString` parameter is a Boolean that when true also parses the query string portion of the URL into an object literal. The default is false.
- The `slashesDenoteHost` is also a Boolean that when true parses a URL with the format of `//host/path` to `{host: 'host', pathname: '/path'}` instead of `{pathname: '//host/path'}`. The default is false.
- You can also convert a URL object into a string form using the following `url.parse()` method
- The following shows an example of parsing a URL string into an object and then converting it back into a string:

```
var url = require('url');  
var urlStr = 'http://user:pass@host.com:80/resource/path?query=string#h  
var urlObj = url.parse(urlStr, true, false);  
urlString = url.format(urlObj);
```


Resolving the URL Components

To resolve a URL to a new location use the following syntax:

url.resolve(from, to)

The **from** parameter specifies the original base URL string. The **to** parameter specifies the new location where you want the URL to resolve. The following code illustrates an example of resolving a URL to a new location.

```
var url = require('url');  
var originalUrl =  
'http://user:pass@host.com:80/resource/path?query=string#hash'  
var newResource = '/another/path?querynew';  
console.log(url.resolve(originalUrl, newResource));
```

The output is shown below.

http://user:pass@host.com:80/another/path?querynew

Processing Query Strings and Form Parameters

- The query string and form parameters are just basic key-value pairs.
- To convert the string into a JavaScript object using the `parse()` method from the `querystring` module:

`querystring.parse(str, [sep], [eq], [options])`

- The `str` parameter is the query or parameter string. The `sep` parameter allows you to specify the separator character used. The default separator character is `&`. The `eq` parameter allows you to specify the assignment character to use when parsing. The default is `=`.
- The `options` parameter is an object with the property `maxKeys`. The default is 1000. If you specify 0, there is no limit.

Understanding Request, Response, and Server Objects

The `http.ClientRequest` Object

- The `ClientRequest` object is created internally **when you call `http.request()`** when building the HTTP client.
- The `ClientRequest` implements a Writable stream, so it provides all the functionality of a Writable stream object.
- To implement a `ClientRequest` object, you use a call to `http.request()`

using the following syntax:

`http.request(options, callback)`

- The `options` parameter is an object whose properties define how to open and send the client HTTP request to the server.
- The `callback` parameter is a callback function that is called after the request is sent to the server and handles the response back from the server.

- **The `http.ServerResponse` Object**

- The `ServerResponse` object is created by the HTTP server internally when a request event is received.
- It is passed to the request event handler as the second argument. You use the `ServerRequest` object to formulate and send a response to the client.
- The `ServerResponse` implements a Writable stream, so it provides all the functionality of a Writable stream object.
- For example, you can use the `write()` method to write to it as well as pipe a `Readable` stream into it to write data back to the client.

The `http.IncomingMessage` Object

- The `IncomingMessage` object is created either by the HTTP server or the HTTP client.
- On the server side, the client request is represented by an `IncomingMessage` object, and on the client side the server response is represented by an `IncomingMessage` object.

- The most basic type of HTTP server is one that serves static files. To serve static files from Node.js, you need to first start the HTTP server and listen on a port.
- Then in the request handler, you open the file locally using the fs module and write the file contents to the response.

Implementing a basic static file webserver

```
var fs = require('fs');
var http = require('http');
var url = require('url');
var ROOT_DIR = "html/";
http.createServer(function (req, res)
{
var urlObj = url.parse(req.url, true, false);
fs.readFile(ROOT_DIR + urlObj.pathname, function (err,data) {
if (err) {
res.writeHead(404);
res.end(JSON.stringify(err));
return;
}
res.writeHead(200);
res.end(data);
});
}).listen(8080);
```

Basic web client retrieving static files

```
var http = require('http');
var options = {
  hostname: 'localhost',
  port: '8080',
  path: '/hello.html'
};
function handleResponse(response) {
  var serverData = "";
  response.on('data', function (chunk) {
    serverData += chunk;
  });
  response.on('end', function () {
    console.log(serverData);
  });
}
http.request(options, function(response){
  handleResponse(response);
}).end();
```


- **Output Implementing a basic static file webserver**

```
C:\books\node\ch07>node http_server_static.js
```

```
<html>
```

```
<head>
```

```
<title>Static Example</title>
```

```
</head>
```

```
<body>
```

```
<h1>Hello from a Static File</h1>
```

```
</body>
```

```
</html>
```



Implementing Dynamic GET Servers

```
var http = require('http');
var messages = [
  'Hello World',
  'From a basic Node.js server',
  'Take Luck'];
http.createServer(function (req, res) {
  res.setHeader("Content-Type", "text/html");
  res.writeHead(200);
  res.write('<html><head><title>Simple HTTP Server</title></head>');
  res.write('<body>');
  for (var idx in messages){
    res.write('\n<h1>' + messages[idx] + '</h1>');
  }
  res.end('\n</body></html>');
}).listen(8080);
```

```
var options = {  
  hostname: 'localhost',  
  port: '8080',};  
function handleResponse(response) {  
  var serverData = "";  
  response.on('data', function (chunk) {  
    serverData += chunk; });  
  response.on('end', function() {  
    console.log("Response Status:", response.statusCode);  
    console.log("Response Headers:", response.headers);  
    console.log(serverData); });  
  }  
  http.request(options, function(response){  
    handleResponse(response);  
  }).end
```

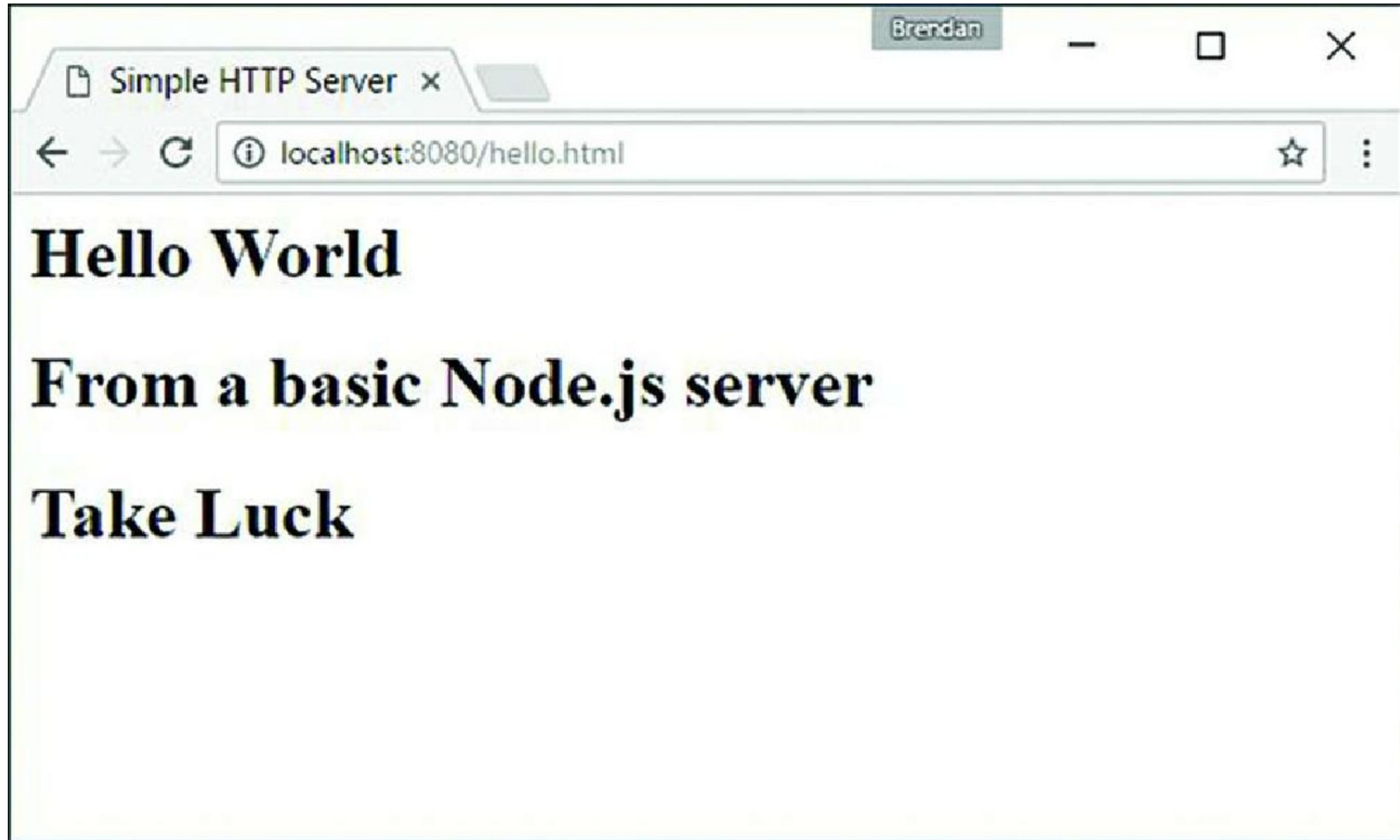
Basic web client that makes a GET request to the server

```
C:\books\node\ch07>node http_server_get.js
```

```
Response Status: 200
```

```
Response Headers: { 'content-type': 'text/html',  
date: 'Mon, 26 Sep 2016 17:10:33 GMT',  
connection: 'close',  
'transfer-encoding': 'chunked' }
```

```
<html><head><title>Simple HTTP Server</title></head><body>  
<h1>Hello World</h1>  
<h1>From a basic Node.js server</h1>  
<h1>Take Luck</h1>  
</body></html>
```



Summary

Node JS

- Node.js is a website/application framework.
- The fact that Node.js is written in JavaScript allows developers to easily navigate back and forth between client and server code.

Installing Node JS

- \$npm install <module name>

Node Packaged Modules

- A Node Packaged Module is a packaged library that can easily be shared, reused, and installed in different projects.

Node Packaged Registry

- The Node modules have a managed location called the Node Package Registry where packages are registered.

Node Package Manager

- It allows you to find, install, remove, publish, and do everything else related to Node Package Modules.
- It provides the link between the Node Package Registry and your development environment.

Creating a Node.js Packaged Module

To create a Node.js Packaged Module,

- create the functionality in JavaScript,
- define the package using a package.json file,
- either publish it to the registry or package it for local use.

Using Events

- Blocking I/O
- Implementing Timers
 - Delaying work with timeouts
 - Interval
 - Immediate

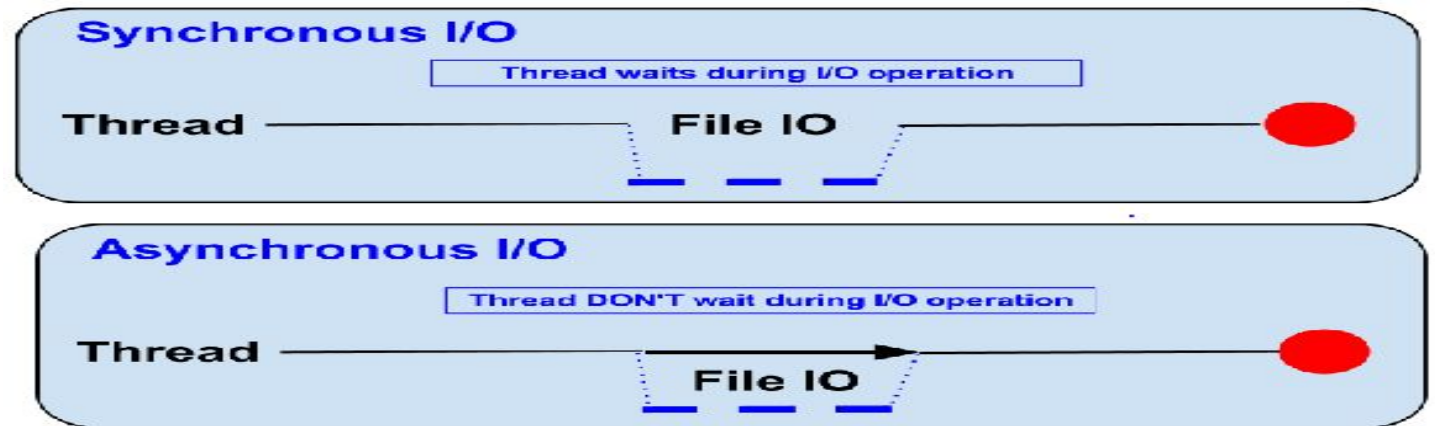
Event Emitter and Listener

Callbacks

- Add additional parameters
- Closure in callbacks
- Chaining callbacks

Handling Data I/O

- Synchronous
- Asynchronous



Implementing HTTP services