1

Making A Web Server

In this chapter, we will cover:

- Setting Up A Router
- Serving Static Files
- Caching Content in Memory For Immediate Delivery
- Optimizing Performance with Streaming
- Securing Against Filesystem Hacking Exploits

Introduction

One of the great qualities of Node is its simplicity. Unlike PHP or ASP there is no separation between web server and code nor do we have to customize large configuration files to get the behavior we want. With Node we can create the server, customize it, and deliver content all at the code level. This chapter demonstrates how to create a web server with Node and feed content through it, whilst implementing security and performance enhancements to cater for various situations.

Setting Up A Router

In order to deliver web content we need to make a URI available. This recipe walks us through the creation of an HTTP server that exposes routes to the user.

Getting ready

First let's create our server file. If our main purpose is to expose server functionality it's general practice to call the file server.js. It's also a good idea to install and use hotnode:

sudo npm -g install hotnode

hotnode server.js

Comment [1]: wavded:
This may be referenced already or assumed knowledge by the reader, but I don't see any directions on how to setup node and npm on the users system (even if it is a reference to a web resource). Another thought is putting this project into a folder. Again may be assumed but wouldn't hurt to keep things organized.
(JT) Although setup is out of this books interest using folders or a more expliite instruction of where to put files maybe of value.

Hotnode will conveniently auto-restart the server when we save changes.

How to do it...

In order to create the server we need the http module, so let's load it and use the http.createServer method:

```
var http = require('http');
http.createServer(function (request, response) {
response.writeHead(200, {'Content-Type': 'text/html;'});
response.end('Woohoo!');
}).listen(8080);
```

Now, if we save our file and access localhost:8080 on a web browser or using curl, our browser (or curl) will exclaim: 'Woohoo!'. But the same will occur at localhost:8080/foo. Indeed, any path will render the same behavior. So let's build in some routing. We can use the path module to extract the basename of the path (the final part of the path), and reverse any URI encoding from the client with decodeURI:

```
var http = require('http');
var path = require('path');
http.createServer(function (request, response) {
var lookup=path.basename(decodeURI(request.url));
```

We now need a way to define our routes. One option is to use an array of objects:

```
var pages = [
{route: '',output:'Woohoo!'},
{route: 'about',output:'A simple routing with Node example'},
{route: 'another page',output:function() {return 'Here\'s
'+this.route;}},
];
```

Our pages array should be placed above the http.createServer call.

Within our server, we need to loop through our array and see if the lookup variable matches any of our routes. If it does we can supply the output. We'll also implement some 404 handling:

```
http.createServer(function (request, response) {
  var lookup=path.basename(decodeURI(request.url));
```

Comment [2]: wavded: semi not needed if no params specified (http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html#sec3.7)

Comment [3]: wavded: Windows users may not be familiar with this command, unless you have a particular reason why to use curl (e.g. viewing headers), it may not necessary to mention. If you do want to use that curl functionality, may be worth mentioning how Windows users can get it installed.

Comment [4]: wavded: res.send(page.output) throws an error in node 0.6.x for 'another page', it does work if you actually are calling the function but in your example you are just accessing the property. I think its a cool example but at its current point does not work as expected.

Comment [5]: wavded: Unneeded comma

```
  pages.forEach(function(page) {
    if (page.route === lookup) {
      response.writeHead(200, {'Content-Type': 'text/html'});
      response.end(page.output);
    }
  });
  if (!response.finished) {
    response.writeHead(404);
    response.end();
  }
}).listen(8080);
```

How it works...

The callback function we provide to http.createServer gives us all the functionality we need to interact with our server through the request and response objects. We use request to obtain the requested URL and then we acquire its basename with path. We also use decodeURI, without which our another page route would fail as our code would try to match "another%20page" against our pages array and return false.

Once we have our basename we can match it in any way we want. We could send it in a database query to retrieve content, use regular expressions to effectuate partial matches, or we could match it to a file name and load its contents.

We could have used a switch statement to handle routing but our pages array has several advantages: it's easier to read, easier to extend and can be seamlessly converted to JSON. We loop through our pages array using forEach.

Node is built on Google's V8 engine, which provides us with a number of ECMAScript 5 features. These features can't be used in all browsers as they're not yet universally implemented, but using them in Node is no problem! forEach is an ES5 implementation; the ES3 way is to use a nasty for loop.

Whilst looping through each object, we check its route property. If we get a match we write the 200 OK status and content-type headers then we end the response with the objects output property.

Response.end allows us to pass a parameter to it, which it writes just before finishing the response. In response.end we have used a ternary operator (?:) to conditionally call

Comment [6]: wavded:
It may be worth having 404 output something to the browser so the user can see they hit a page without a route.

Comment [7]: I have broken this up a bit as it was a lot of information to read all at once. You may want to break it up differently, or break it up a little bit more.
SarahC

Comment [8]: wavded:
nasty :) its not that bad, just not as convenient. you may need to describe what you mean or just accent the new way without making reference to the old.

page.output as a function or simply pass it as a string. Notice the another page route contains a function instead of a string. The function has access to its parent object through the this variable, and allows for greater flexibility in assembling the output we want to provide. In the event that there is no match in our forEach loop response.end would never be called and therefore the client would continue to wait for a response until it times out. To avoid this, we check the response.finished property and if it's false we write a 404 header and end the response.

Response.finished depends on the forEach callback yet it's not nested within the callback. On the surface this looks like a potential race condition, however forEach does not operate asynchronously; it blocks until all loops are complete.

Comment [9]: I have broken this up a bit as it was a lot of information to read all at once. You may want to break it up differently, or break it up a little bit more.
SarahC

Comment [10]: wavded:
I wonder if users would think this way, if its someone coming from most other server-side langs, I think they would assume this is blocking already.. or not.

There's more...

There are many ways to extend and alter this example, there's also some great non-core modules available that do the leg work for us.

Simple Multilevel Routing

Our routing so far only deals with a single level path, a multilevel path (e.g. "/about/node") will simply return a 404. We can alter our object to reflect a subdirectory like structure, remove path and use request.url for our routes instead of path.basename.

```
Var http=require('http');
var pages = [
{route: '/',output:'Woohoo!'},
{route: '/about/this',output:'Multilevel routing with Node'},
{route:'/about/node',output:'Evented I/O for V8 JavaScript.'},
{route: '/another page',output:function() {return 'Here\'s '+this.route;}}
];
http.createServer(function (request, response) {
  var lookup=decodeURI(request.url);
```

Warning: Don't pass request.url directly for static file hosting! Check out **Securing Against Filesystem Hacking Exploits**.

Comment [11]: wavded:
It isn't clear to me what we are passing request.uri directly to in what you've mentioned above, (assigning to lookup variable)?

Multilevel routing could be taken further, we could build and then traverse a more complex object,

```
{route:'about',childRoutes:[
                    {route:'node',output:'Evented I/O for V8
Javascript'},
                    {route:'this',output:'Complex Multilevel Example'}
                   ]}
```

After the third or~~of~~ fourth level this object would become a leviathan to look at. We could alternatively create a helper function to define our routes that essentially pieces our object together for us. Alternatively we could use one of the excellent non-core routing modules provided by the open source Node community. Excellent solutions already exist that provide helper methods to handle the increasing complexity of scalable multilevel routing.(see **Routing Modules** and **Chapter 6: Accelerating Development with Express**).

Parsing The Querystring

Two other useful core modules are url and querystring. The url.parse method allows two parameters, first the URL string (in our case, this will be request.url) and second a boolean parameter named parseQueryString. If set to true, it lazy loads the querystring module (saving us the need to require it) to parse the query into an object. This makes it easy for us to interact with the query portion of a URL.

```
var http = require('http');
var url = require('url');
var pages = [
{id:'1',route: '',output:'Woohoo!'},
{id:'2',route: 'about',output:'A simple routing with Node example'},
{id:'3',route: 'another page',output:function() {return this.route;}},
];
http.createServer(function (request, response) {
  var id=url.parse(decodeURI(request.url),true).query.id;
  if (id) {
    pages.forEach(function(page) {
      if (page.id === id) {
       response.writeHead(200, {'Content-Type': 'text/html'});
       response.end(typeof page.output === 'function'
```

```
                    ? page.output() : page.output);
    }
  });
 }
 if (!response.finished) {
   response.writeHead(404);
   response.end();
 }
}).listen(8080);
```

With the added id properties we can access our object data by, for instance localhost:8080?id=2.

Comment [12]: wavded:
this is what was missing from the first example, the 'function' route would fail.

Routing Modules

There's an up to date list of various routing modules for node at https://github.com/joyent/node/wiki/modules#wiki-web-frameworks-routers. These community made routers cater to various scenarios. It's important to research the activity and maturity of a module before taking it into a production environment. In **Chapter 6 Accelerating Development with Express** we will go into greater detail on using the built in Express/Connect router for more comprehensive routing solutions.

See also

- Making A Web Server: Serving Static Files
- Accelerating Development with Express : Dynamic Routing
- Securing Against Filesystem Hacking Exploits

Serving Static Files

If we have information stored on disk that we want to serve as web content we can use the the fs (filesystem) module to load our content and pass it through the createServer callback. This is a basic conceptual starting point for serving static files: as we will learn in the following recipes there are much more efficient solutions.

Getting ready

We'll need some files to serve. Let's create a directory named content, containing the following 3 files: ~~and with three files named index.html, styles.css and script.js. For the HTML:~~

index.html:

```
<html>
<head>
<title>Yay Node!</title>
<link rel=stylesheet href=styles.js type=text/css>
<script src=script.js type=text/javascript></script>
</head>
<body>
<span id=yay>Yay!</span>
</body>
</html>
```

~~And the Javascript:~~

script.js

```
window.onload=function() {alert('Yay Node!');};
```

~~And the CSS:~~

styles.css:

```
#yay {font-size:5em;background:blue;color:yellow;padding:0.5em}
```

How to do it...

As in the previous recipe, we'll be using the core modules http and path. We'll also need to access the file system, so we'll require fs too. Let's create the server and use the path module to check if a file exists.

```
var http = require('http');
var path = require('path');
var fs = require('fs');
http.createServer(function (request, response) {
  var lookup=path.basename(decodeURI(request.url)) || 'index.html';
```


Comment [13]: You don't need quotation marks if you are using the Code in Text style.
SarahC

Comment [14]: You don't need quotation marks if you are using the Code in Text style.
SarahC

Formatted: Strikethrough

Formatted: Strikethrough

Comment [15]: So is this what we put in index.html?
SarahC

Comment [16]: So is this what we put in index.html?
SarahC

Comment [17]: wavded:
Typo: should be styles.css

Comment [18]: The script.js file?
SarahC

Comment [19]: The script.js file?
SarahC

Formatted: Strikethrough

Formatted: Strikethrough

Formatted: Strikethrough

Comment [20]: Styles.css file??
Make it all crystal clear so no mistakes can be made ;)

Very good that you've provided example content though.
SarahC

Comment [21]: Styles.css file??
Make it all crystal clear so no mistakes can be made ;)

Very good that you've provided example content though.
SarahC

Formatted: Strikethrough

Formatted: Font: Verdana, 9 pt, Font color: Red

Comment [JT22]: The changes here were made by one of the reviewers, there is no content difference so the style is entirely upto you, please feel free to change it back if you wish...

```
  var f='content/'+lookup;
  path.exists(f, function (exists) {
    console.log(exists ? lookup+" is there" : lookup+" doesn't exist");
  });
}).listen(8080);
```

If we haven't already, we can initialize our server.js file :

hotnode server.js

Try loading localhost:8080/foo, the console will say “foo doesn't exist”, because it doesn't. localhost:8080/script.js will tell us “script.js is there”, because it is. Before we can serve a file we are supposed to let the client know the content-type, which we can determine from the file extensions. So let's make a quick map using an object:

```
var mimeTypes={
'.js' : 'text/javascript',
'.html': 'text/html',
'.css' : 'text/css'
};
```

We could extend our mimeTypes map later to support more types.

> Modern browsers may be able to interpret certain mime types (like text/javascript) without the server sending a content-type header, but older browsers or less common mime types will rely upon the correct content-type header being sent from the server.

Remember to place mimeTypes outside of the server callback~~callback~~, we don't want to initialize the same object on every client request. If the requested file exists we can convert our file extension into a content-type by feeding path.extname into mimeTypes, then pass our retrieved content-type to response.writeHead. If the requested file doesn't exist, we'll write out a 404 and end the response.

```
//requires variables, mimeType object...
http.createServer(function (request, response) {
  var lookup=path.basename(decodeURI(request.url)) || 'index.html';
  var f='content/'+lookup;
  path.exists(f, function (exists) {
```

```
    if (exists) {
var contentType={'Content-type':mimeTypes[path.extname(lookup)]};
      response.writeHead(200,contentType);
      response.end();
      return;
    }
      response.writeHead(404);
      response.end();
  });
}).listen(8080);
```

Comment [23]: wavded: writeHead takes an object of http headers of which one is could be 'content-type', do you think its confusing to have the object be named contentType instead of like 'headers' and passed to the function? maybe or maybe not. just was thinking about it so thought i'd mention.

Comment [24]: wavded: early return here doesn't seem to gain you much from a readability standpoint, a simple else would be easier to read IMO.

At the moment there is still no content sent to the client. We have to get this content from our file, so we wrap the response handling in an fs.readFile method callback.

```
//http.createServer, inside path.exists:
if (exists) {
  fs.readFile(f,function(err,data) {
    var contentType={'Content-type':mimeTypes[path.extname(lookup)]};
    response.writeHead(200,contentType);
    response.end(data);
  });
 return;
}
```

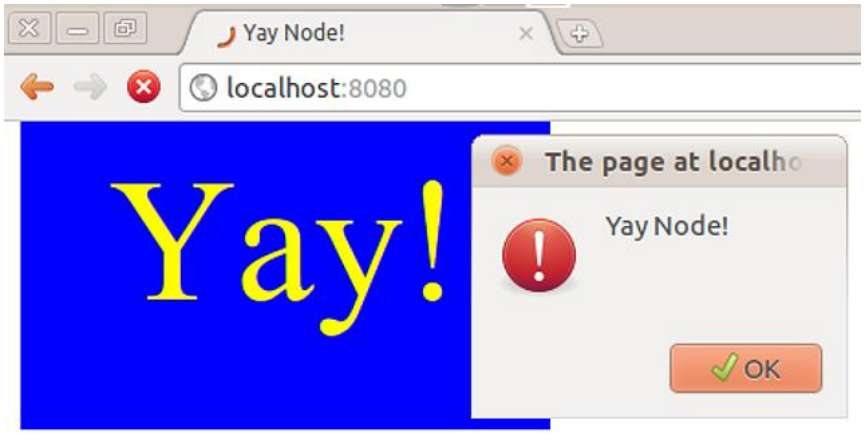
Before we finish, let's apply some error handling to our fs.readFile callback

```
//requires variables, mimeType object...
//http.createServer,  path exists, inside if(exists):
fs.readFile(f,function(err,data) {
    if (err) {response.writeHead(500);response.end(); return;}
    var contentType={'Content-type':mimeTypes[path.extname(lookup)]};
    response.writeHead(200,contentType);
    response.end(data);
  });
```

```
    return;
}
```

Notice return stays outside of the fs.readFile callback. We are returning from the path.exists callback to prevent further code execution (e.g. sending the 404). Placing a return in an if statement is the equivalent of using an else branch. However the if return pattern is generally preferable to using if else in Node, as it eliminates yet another set of curly braces.

So now we can navigate to localhost:8080 which will serve our index.html file. The index.html file makes calls to our script.js and styles.css files which our server also delivers with appropriate mime types. The result:



7188_01_01.png

This recipe serves to illustrate the fundamentals of serving static files. Remember, this is **not** an efficient solution! In a real world situation we don't want to make an I/O call every time a request hits the server, this is very costly especially with larger files.

How it works...

Our script creates a server and declares a variable called lookup. We assign a value to lookup using the double pipe (||) "or" operator. This defines a default route if path.basename is empty. Then we pass lookup to a new variable that we named~~named~~ f

Comment [25]: joe.d.developer: Perhaps touch on where to find better approaches? Such as that this will be approached in more depth later in the Chapter.

JoeDev

Comment [26]: wavded: why not file or something more descriptive than just f? Would help me from asking the question, what was f again?

in order to prepend our content directory to the intended filename. Next we run f through the path.exists method and check the exist parameter in our callback to see if the file is there. If the file does exist we read it asynchronously using fs.readFile. If there is a problem accessing the file we write a 500 server error, end the response and return from the fs.readFile callback. We can test the error handling functionality by removing read permissions from index.html.

chmod -r index.html

Doing so will cause the server to throw the 500 server error status code. To set things right again:

chmod +r index.html

As long as we can access the file we grab the content-type using our handy mimeTypes mapping object, write the headers, end the response with data loaded from the file and finally return from the function. If the requested file does not exist we bypass all this logic, write a 404 and end the response.

There's more...

Here's something to watch out for...

The Favicon Gotcha

When using a browser to test our server sometimes an unexpected server hit can be observed. This is the browser requesting the default favicon.ico icon file that servers can provide. Apart from the initial confusion of seeing additional hits this is usually not a problem. If the favicon request does begin to interfere we can handle it thusly:

```
if (request.url==='/favicon.ico') {
  response.end();
  return;
}
```

If we wanted to be more polite to the client, we could also inform it of a 404 by using response.writeHead(404) before issuing response.end.

See also

- Caching Content in Memory For Immediate Delivery
- Optimizing Performance with Streaming
- Securing Against Filesystem Hacking Exploits

Caching content in memory for immediate delivery

Comment [27]: Please use sentence case for headings. Only the main title needs title case.
SarahC

Directly accessing storage on each client request is not the ideal. For this task we will explore how to enhance server efficiency by accessing the disk only on the first request, caching the data from file for that first request and serving all further requests out of the process memory.

Getting ready

We are going to improve upon the code from the previous task, so we'll be working with server.js and in the content directory with index.html, styles.css and script.js.

How to do it...

Let's begin by looking at our script from the previous recipe **Serving Static Files**:

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

var mimeTypes={
'.js' : 'text/javascript',
'.html': 'text/html',
'.css' : 'text/css'
} ;

http.createServer(function (request, response) {
  var lookup=path.basename(decodeURI(request.url)) || 'index.html';
  var f='content/'+lookup;
  path.exists(f, function (exists) {
    if (exists) {
      fs.readFile(f,function(err,data) {
      if (err) {response.writeHead(500);response.end(); return;}
      var contentType={'Content-
type':mimeTypes[path.extname(lookup)]};
```

```
        response.writeHead(200,contentType);
        response.end(data);
      });
      return;
    }
      response.writeHead(404); //no such file found!
      response.end();
  });
```

We need to modify this code to only read the file once, load its contents into memory and afterwards respond to all requests for that file from memory. To keep things simple and preserve maintainability, we'll extract our cache handling and content delivery into a separate function. So above http.createServer, and below mimeTypes we'll add the following:

```
var cache={};
function cacheAndDeliver(f,cb) {
  if (!cache[f]) {
    fs.readFile(f,function(err,data) {
      if (!err) {
        cache[f] = {content:data} ;
      }
      cb(err,data);
    });
    return;
  }
  console.log('loading '+f+' from cache');
  cb(null,cache[f].content);
}
//http.createServer …..
```

A new cache object has been added to store our files in memory as well as a new function called cacheAndDeliver. Our function takes the same parameters as fs.readFile

Comment [28]: wavded:
Don't think this early return is as easily readable/understandable. I would write it this way:

```
function cacheAndDeliver(f,cb) {
  if (cache[f]) {
    // truthy condition first
  }
  else {
    // fetch file from system
  }
}
```

so we can replace fs.readFile in the http.createServer callback whilst leaving the rest of the code intact:

```
//...inside http.createServer:
path.exists(f, function (exists) {
  if (exists) {
    cacheAndDeliver(f,function(err,data) {
      if (err) {response.writeHead(500);response.end(); return;}
      var contentType={'Content-type':mimeTypes[path.extname(f)]};
      response.writeHead(200,contentType);
      response.end(data);
    });
   return;
  }
//rest of path exists code (404 handling)...
```

When we execute our server.js file and access localhost:8080 twice consecutively the second request causes the console to output:

```
loading content/index.html from cache
loading content/styles.css from cache
loading content/script.js from cache
```

How it works...

We defined a function called cacheAndDeliver which like fs.readFile takes a filename and callback as parameters. This is great because we can pass the exact same callback of fs.readFile to cacheAndDeliver padding the server out with caching logic without adding any extra complexity visually to the inside of the http.createServer callback. As it stands, the worth of abstracting our caching logic into an external function is arguable, but the more we build on the servers caching abilities the more feasible and useful this abstraction becomes. Our cacheAndDeliver function checks to see if the requested content is already cached, if not we call fs.readFile and load the data from disk. Once we have this data we may as well hold onto it, so it's placed into the cache object referenced by its file path (the f variable). The next time anyone requests the file, cacheAndDeliver will see that we have the file stored in the cache object and will issue an alternative callback containing the cached data. Notice we fill the cache[f] property

with another new object containing a content property. This makes it easier to extend the caching functionality in the future as we would just have to place extra properties into our cache[f] object and supply logic that interfaces with these properties accordingly.

There's more...

If we were to modify the files we are serving, any changes wouldn't be reflected until we restart the server. We can do something about that.

Reflecting Content Changes

To detect whether a requested file has changed since we last cached it we must know when the file was cached and when it was last modified. To record when the file was last cached, let's extend the cache[f] object:

```
cache[f] = {content:data,
            timestamp: Date.now() //store a Unix time stamp
           };
```

That was easy, now to find out when the file was updated last. The fs.stat method returns an object as the second parameter of its callback. This object contains the same useful information as the command line GNU coreutils stat. fs.stat supplies three time related properties: last accessed (atime), last modified (mtime) and last changed (ctime). The difference between mtime and ctime is that ctime will reflect *any* alterations to the file, whereas mtime will only reflect alterations to the *content* of the file. Consequently if we changed the permissions of a file ctime would update but mtime would stay the same. We want to pay attention to permission changes as they happen so let's use the ctime property:

```
//requires and mimeType object....
var cache={};
function cacheAndDeliver(f,cb) {
  fs.stat(f,function(err,stats) {
    var lastChanged=Date.parse(stats.ctime);
    var isUpdated=(cache[f]) && lastChanged  > cache[f].timestamp;
    if (!cache[f] || isUpdated) {
      fs.readFile(f,function(err,data) {
        console.log('loading '+f+' from file');
```

```
    //rest of cacheAndDeliver...
  }); // end of fs.stat
  } // end of cacheAndDeliver
```

The contents of cacheAndDeliver has been wrapped in an fs.stat callback, two variables have been added and the if(!cache[f]) statement has been modified. We parse the ctime property of the second parameter dubbed stats using Date.parse to convert it to Unix time (milliseconds since the Unix epoch) and assign it to our lastChanged variable. Then we check whether the requested files last changed time is greater than when we cached the file (providing the file is indeed cached) and assign the result to our isUpdated variable. After that its merely a case of adding the isUpdated boolean to the conditional if(!cache[f]) statement via the || (or) operator. Whether the file isn't cached **or** if the file is newer than that which is cached, we load from disk and fill (or refill) the cache.

Comment [29]: wavded:
I think milliseconds needs to accented more than that its Unix time, that may not mean much to someone on Windows. The important part is that Date.now and Date.parse are giving the same millisecond representation of the date in order to compare properly. 2 cents. I'm totally a Unix guy so I get what you are saying but thinking about the Windows guy.

Comment [30]: This sentence is a bit difficult to understand. Can you please rephrase it?
SarahC

See also

- Optimizing Performance with Streaming
- Creating a JSON REST Server

Optimizing Performance with Streaming

Caching content certainly improves upon reading a file from disk for every request. However, with fs.readFile we are reading the whole file into memory before sending it out in a responseresponse. For better performance we can stream a file from disk and pipe it directly to the response object, sending data straight to the network socket a piece at a time.

Getting ready

We are building on our code from the last example, so let's get server.js, index.html, styles.css and script.js ready.

How to do it...

We will be using fs.createReadStream to initialize a stream which can be piped to the response object. fs.createReadStream creates a readStream which inherits from the EventEmitter class.

The EventEmitter class isis responsible for the “evented” part in Node's tag line: “Evented I/O for V8 JavaScript”. Due to this we'll be using listeners instead of callbacks

Comment [32]: wavded:
May be a little stretch to say EventEmitter is responsible, libev/libuv is really responsible when it comes to I/O.

to control the flow of stream logic. In this case implementing fs.createReadStream within our cacheAndDeliver function isn't ideal because the event listeners of fs.createReadStream will need to interface with the request and response objects, which for the sake of simplicity would preferably be dealt with in the http.createServer callback. Thus, for brevity we will discard our cacheAndDeliver function and implement basic caching within the server callback:

```
//requires, mime types, createServer, lookup and f vars...
path.exists(f, function (exists) {
  if (exists) {
    var contentType={'Content-type':mimeTypes[path.extname(f)]};
    if (cache[f]) {
      response.writeHead(200,contentType);
      response.end(cache[f].content);
      return;
    } //...rest of server code...
```

Later on, we will fill cache[f].content whilst we're interfacing with the readStream object. Here's how we use fs.createReadStream:

```
var s=fs.createReadStream(f);
```

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

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

Because createReadStream returns the readStream object, we can latch our event listener straight onto it using method chaining with dot notation. Each stream is only going to open once, we don't need to keep on listening to it. Therefore, we can use the once method instead of on to automatically stop listening after the first event occurrence:

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

Comment [31]: This seems like a lot of detail for this section. Could some of it move to the 'How it works' section?

Also, remember to break up long paragraphs.
SarahC

Before we fill out the open event callback, let's implement some error handling

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

The key to this whole endeavor is the stream.pipe method. This is what enables us to take our file straight from disk and stream it directly to the network socket via our response object.

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

But what about ending the response? Conveniently, stream.pipe detects when the stream has ended and calls the response.end() for us. There's one other event we need to listen to, for caching purposes.

```
fs.stat(f,function(err,stats) {
        var bufferOffset=0;
        cache[f] = {content:new Buffer(stats.size)};
        s.on('data',function(chunk) {
        chunk.copy(cache[f].content,bufferOffset);
        bufferOffset+=chunk.length;
    });
  });
```

Comment [33]: wavded:
It isn't clear where this bit of code goes from its surrounding context.

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

How it works...

Instead of the client waiting for the server to load the entire file from disk prior to sending it to the client, we use a stream to load the file in small ordered pieces and promptly send them to the client. With larger files this is especially useful, as there is minimal delay between the file being requested and the client starting to receive the file. We did this by using fs.createReadStream to start streaming our file from disk. Then we added an open event listener using the once method since we want to stop listening for open once it has triggered. We respond to the open event by writing the headers and using the stream.pipe method to shuffle the incoming data straight to the client. stream.pipe handles the data flow. If the client becomes overwhelmed with processing it sends a signal to the server which should be honored by pausing the stream. Under the hood stream.pipe uses stream.pause and stream.resume to manage this interplay. Whilst the response is being piped to the client, the content cache is simultaneously being filled. To achieve this we had to create an instance of the Buffer class for our cache[f].content property. A Buffer must be supplied with a size (or array, or string) which in our case is the size of the file. To get the size we used the asynchronous fs.stat and captured the size property in the callback. The data event returns a Buffer as its only callback parameter. The default bufferSize for a stream is 64kb, any file whose size is less than the bufferSize will only trigger one data event because the whole file will fit into the first chunk of data. But for files that are greater than the bufferSize we have to fill our cache[f].content property a piece at a time.

Comment [34]: Please break up this paragraph.
SarahC

Changing the Default readStream Buffer Size

We can change the buffer size of our readStream by passing an options object with a bufferSize property as the second parameter of fs.createReadStream.

For instance to double the buffer you could use
fs.createReadStream(f,{bufferSize: 128 * 1024});

We cannot simply concatenate each chunk with cache[f].content because this will coerce binary data into string format which, though no longer in binary format, will later be interpreted as binary. Instead we have to copy all the little binary buffer chunks into our binary cache[f].content buffer.

Comment [35]: wavded:
Seems bad for perf as well.

We created a bufferOffset variable to assist us with this. Each time we add another chunk to our cache[f].content buffer we update our new bufferOffset by adding the

length of the chunk buffer to it. When we call the Buffer.copy method on the chunk buffer we pass bufferOffset as the second parameter so our cache[f].content buffer is filled correctly.

Moreover, operating with the Buffer class renders performance enhancements with larger files because it bypasses the V8 garbage collection methods which tend to fragment large amounts of data thus slowing down Node's ability to process them.

There's more...

Whilst streaming has solved a problem of waiting for files to load into memory before delivering them, we are nevertheless still loading files into memory via our cache object. With larger files, or large amounts of files, this could have potential ramifications.

Protecting Against Process Memory Overruns

There is a limited amount of process memory. By default V8's memory is set to 1400mb on 64bit systems and 700mb on 32bit. This can be altered by running node with --max-old-space-size=N where N is the amount of megabytes (the actual maximum amount that it can be set to depends upon the OS and of course the amount of physical RAM available). If we absolutely had to be memory intensive we could run our server on a large cloud platform, divide up the logic and start new instances of node using the child_process class.

In this case, high memory usage isn't necessarily required and we can optimize our code to significantly reduce the potential for memory overruns. There is less benefit to caching larger files, because the slight speed improvement relative to the total download time is negligible whilst the cost of caching them is quite significant in ratio to our available process memory. We can also improve cache efficiency by implementing an expiration time on cache objects that can then be used to clean the cache, consequently removing files in low demand and prioritizing high demand files for faster delivery. Let's rearrange our cache object slightly:

```
var cache={
store: {},
maxSize : 26214400, //(bytes) 25mb
}
```

For a clearer mental model we're making a distinction between the cache as a functioning entity and the cache as a store (which is a part of the broader cache entity). Our first goal is to only cache files under a certain size, we've defined cache.maxSize for this purpose. All we have to do now is insert an if condition within the fs.stat callback:

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

Notice we also slipped in a new timestamp property into our cache.store[f], this is for our second goal: cleaning the cache. Let's extend cache:

```
var cache={
store: {},
maxSize : 26214400, //(bytes) 25mb
maxAge : 5400*1000, //(ms) 1 and a half hours
clean: function(now) {
  that=this;
    Object.keys(this.store).forEach(function(file) {
      if (now > that.store[file].timestamp+that.maxAge) {
        delete that.store[file];
      }
    });
}
};
```

So in addition to maxSize, we've created a maxAge property and added a clean method. We call cache.clean at the bottom of the server, like so:

```
//all of our code prior
```

```
  cache.clean(Date.now());
}).listen(8080); //end of the http.createServer
```

cache.clean loops through the cache.store and checks to see if it has exceeded its specified lifetime, if it has we remove it from the store. One further improvement and then we're done. cache.clean is called on each request. This means the cache.store is going to be looped through on every server hit, which is neither necessary nor efficient. It would be better if we cleaned the cache, say, every two hours or so. We'll add two more properties to cache: cleanAfter to specify how long between cache cleans, and cleanedAt to determine how long it has been since the cache was last cleaned.

```
var cache={
store: {},
maxSize : 26214400, //(bytes) 25mb
maxAge : 5400*1000, //(ms) 1 and a half hours
cleanAfter:7200 * 1000,//(ms) two hours
cleanedAt: 0, //to be set dynamically
clean: function(now) {
 if (now-this.cleanAfter > this.cleanedAt) {
   this.cleanedAt = now;
   that=this;
    Object.keys(this.store).forEach(function(file) {
      if (now > that.store[file].timestamp+that.maxAge) {
        delete that.store[file];
      }
    });
 }
}
};
```

So we wrap our cache.clean method in an if statement that will allow a loop through cache.store *only* if it has been longer than two hours (or whatever cleanAfter is set to), since the last clean.

Comment [36]: wavded:
Just a thought, it would be interesting for me to see benchmarks (like ab) for each of these methods to demonstrate perf diffs with each method. (fs.readFile w/ no cache, basic cache w/ fs.readFile, file streaming, streaming cache)

See also

- Handling File Uploads as a Stream
- Securing Against Filesystem Hacking Exploits

Securing Against Filesystem Hacking Exploits

For a Node app to be insecure there must be something an attacker can interact with for exploitation purposes. Due to Node's minimalist approach the onus is on the programmer to ensure that their implementation doesn't expose security flaws. This recipe will help identify some security risk anti-patterns that could occur when working with the filesystem.

Getting ready

We'll be working with the same content directory as in previous recipes, but we'll start a new insecure_server.js file (there's a clue in the name!) from scratch to demonstrate mistaken techniques.

How to do it...

Our previous static file recipes tend to use path.basename to acquire a route, but this flat levels all request. If we accessed localhost:8080/foo/bar/styles.css our code would take styles.css as the basename and deliver content/styles.css to us. How about we make a subdirectory in our content folder call it subcontent and move our script,js and styles.css files into it. We'd have to alter our script and link tags in index.html:

```
<link rel=stylesheet type=text/css href=subcontent/styles.css>

<script src=subcontent/script.js type=text/javascript></script>
```

We can use the url module to grab the entire pathname. So let's include the url module in our new insecure_server.js file, create our HTTP server and use pathname to get the whole requested path:

```
var http = require('http');
var path = require('path');
var url = require('url');
var fs = require('fs');
http.createServer(function (request, response) {
  var lookup=url.parse(decodeURI(request.url)).pathname;
```

```
    lookup = (lookup === "/") ? '/index.html' : lookup;
    var f='content'+lookup;
    console.log(f);
     fs.readFile(f, function(err, data) {
       response.end(data);
     });
}).listen(8080);
```

If we navigate to localhost:8080, everything works great. We've gone multilevel, hooray. For demonstration purposes a few things have been stripped out from previous recipes (such as path.exists), but even with them this code presents the same security hazards:

curl localhost:8080/../insecure_server.js

Now we have our server's code. An attacker could also access /etc/passwd with a few attempts at guessing its relative path:

curl localhost:8080/../../../../../../../etc/passwd

In order to test these attacks we have to use curl or another equivalent because modern browsers will filter these sorts of requests. As a solution, what if we added a unique suffix to each file we wanted to serve and made it mandatory for the suffix to exist before the server coughs it up? That way, an attacker could request /etc/passwd or our insecure_server.js because they wouldn't have the unique suffix. To try this let's copy the content folder and call it content-pseudosafe, and rename our files to index.html-serve, script.js-serve and styles.css-serve. Let's create a new server file and name it pseudosafe_server.js. Now all we have to do is make the -serve suffix mandatory:

```
//requires section...
http.createServer(function (request, response) {
  var lookup=url.parse(decodeURI(request.url)).pathname;
  lookup = (lookup === "/") ? '/index.html-serve' : lookup+'-serve';
  var f='content-pseudosafe'+lookup;
```

For feedback purposes we'll also include some 404 handling with the help of path.exists

```
//requires, create server etc
path.exists(f, function (exists) {
```

```
    if (!exists) {
      response.writeHead(404);
      response.end();
      return;
    }
  //read file etc
```

So let's start our pseudosafe_server.js file and try out the same exploit:

curl -i localhost:8080/../insecure_server.js

We've used the -i argument so that curl will output the headers. The result? A 404, because the file its actually looking for is ../insecure_server.js-serve, which doesn't exist. So what's wrong with this method? Well it's inconvenient and prone to error. But more importantly an attacker can still work around it!

curl localhost:8080/../insecure_server.js%00/index.html

And voila! There's our server code again. The solution to our problem is path.normalize, which cleans up our pathname before it gets to fs.readFile.

```
http.createServer(function (request, response) {
  var lookup=url.parse(decodeURI(request.url)).pathname;
  lookup = path.normalize(lookup);
  lookup = (lookup === "/") ? '/index.html' : lookup;
  var f='content'+lookup
```

Prior recipes haven't used path.normalize and yet they're still relatively safe. path.basename, gives us the last part of the path, so any leading relative directory pointers (../) are discarded, thus preventing the directory traversal exploit.

How it works...

Here we have two file system exploitation techniques: the relative directory traversal and poison null byte attacks. These attacks can take different forms, such as in a POST request or from an external file. They can have different effects: if we were writing to files instead of reading them an attacker could potentially start making changes to our server. The key to security in all cases is to validate and clean any data that comes from the user. In insecure_server.js we pass whatever the user requests to our fs.readFile method. This is foolish because it allows an attacker to take advantage of the relative

path functionality in our operating system by using ../, thus gaining access to areas that should be off limits. By adding the -serve suffix we didn't solve the problem, we put a plaster on it which can be circumvented by the poison null byte. The key to this attack is the %00 which is a URL hex code for the null byte. In this case the null byte blinds Node to the ../insecure_server.js portion, but when the same null byte is sent through to our fs.readFile method it has to interface with the kernel. But the kernel gets blinded to the index.html part. So our code sees index.html but the read operation sees ../insecure_server.js. This is known as null byte poisoning. To protect ourselves we could use a regex statement to remove the ../ parts of the path, we could also check for the null byte and spit out a 400 Bad Request statement. But we don't have to, because path.normalize filters out the null byte and relative parts for us.

There's more...

White-listing

If security was an extreme priority, we could adopt a strict whitelisting approach. In this approach we would create a manual route for each file we are willing to deliver. Anything not on our whitelist would return a 404. We can place a whitelist array above http.createServer:

```
var whitelist=[
'/index.html',
'/subcontent/styles.css',
'/subcontent/script.js'
];
```

And inside our http.createServer callback we'll put an if statement to check if the requested path is in the whitelist array:

```
if(whitelist.indexOf(lookup) === -1) {
  response.writeHead(404);
  response.end();
  return;
}
```

And that's it, we can test this by placing a file non-whitelisted.html in our content directory.

curl -i localhost:8080/non-whitelisted.html

This will return a 404 because non-whitelisted.html isn't on the whitelist.

Node Static

https://github.com/joyent/node/wiki/modules#wiki-web-frameworks-static has a list of static file server modules available for different purposes. It's a good idea to ensure that a project is mature and active before relying upon it to serve your content. Node-static is a well developed module with built in caching. It's also compliant with the RFC2616 HTTP standards specification which defines how files should be delivered over HTTP. Node-static implements all the essentials discussed in this chapter and more besides. This piece of code is slightly adapted from the node-static Github page at https://github.com/cloudhead/node-static:

```
var static = require('node-static');
var fileServer = new static.Server('./content');
require('http').createServer(function (request, response) {
   request.addListener('end', function () {
      fileServer.serve(request, response);
   });
}).listen(8080);
```

This code will interface with the node-static module to handle server side and client side caching, use streams to deliver content and filter out relative requests and null bytes, among other things.

See also

- Guarding Against HTTP and Socket Exploitation Attacks
- Setting Up an HTTPS Web Server
- Using a Dedicated Server Or VPS