Chapter No. 4
"Response From the Server"
In this package, you will find:
A Biography of the author of the book
A preview chapter from the book, Chapter NO.4 "Response From the Server"
A synopsis of the book’s content
Information on where to buy this book

About the Author

**Hage Yaapa** is a contributing developer of Express and the head of Web and Node.js development at Sourcebits.

He joined one of the best medical schools in India, JIPMER, to become a doctor, but dropped out to pursue his burning passion for computers and the Internet. He has been creating websites and apps since 1999 using a very wide array of web technologies. He is a self-taught programmer and everything he knows about technology, he learned on his own from the Internet and books.

Yaapa blogs about Node.js, Express, and other web technologies on his website www.hacksparrow.com, as Captain Hack Sparrow.
This will sound crazy, but I would like to first thank Khaled Mardam-Bey, the creator of mIRC—the IRC software, which started everything for me. I learned many of the important things I know about computers and the Internet on IRC channels during the transitioning of the century. The first programming language I learned was mIRC Script, which helped me pick up JavaScript and other programming languages rather easily.

Next, I would like to thank T. J. Holowaychuk for creating Express and the unbelievable number of impressive Node.js packages he has created.

Then, I would like to thank Ryan Dahl for creating Node.js, and Isaac Schlueter for carrying it forward.

I would also like to thank Brendan Eich for creating JavaScript, for there would have been no Node.js or Express, if there were no JavaScript.

Last but not the least, I would like to thank my loving wife Kenyum for putting up with me while I wrote this book.

Express Web Application Development

This book is about Express, the popular web framework used by thousands of Node.js developers around the world. It specifically covers the third major version of the framework, commonly referred to as Express 3.

Express has matured considerably since it was first released exactly four years ago. Today it is recognized as one of the best web frameworks for Node.js. Every day new developers from varied backgrounds and experience come to Express for developing their web apps. With its ever-growing popularity, it is about time we had a book on Express.

I wrote a tutorial on Express some time ago that became quite popular online, particularly with those new to Node.js and Express. Ever since, I had a dream of writing a book on Express, which would make no assumptions about the reader's prior experience and knowledge, and still be full of technical details wherever required. The book you are holding in your hands is that dream realized—a book on Express that is both beginner-friendly and technically deep at the same time.

This book covers everything a developer requires to get into serious web development using Express.

What This Book Covers

Chapter 1, *What is Express?*, is a beginner-friendly but technically solid introduction to Express and relevant topics for a strong base right at the start.

Chapter 2, *Your First Express App*, is a practical introduction to building an Express app covering the basics that form the basis of every Express app.

Chapter 3, *Understanding Express Routes*, explains routes in Express in great detail.

Chapter 4, *Response From the Server*, covers the various ways an Express app can respond to a request.

Chapter 5, *The Jade Templating Language*, covers the Jade syntax and its programming capabilities.

Chapter 6, *The Stylus CSS Preprocessor*, covers the Stylus syntax and its programming capabilities.

Chapter 7, *Forms, Cookies, and Sessions*, covers how to handle forms, and create cookies and sessions.

Chapter 8, *Express in Production*, covers important areas to make Express apps production-ready.

For More Information:

www.packtpub.com/express-web-application-development/book
This chapter is about understanding how web servers respond to HTTP requests and how it works in Express. We will cover the details of the response process and learn how to serve different kinds of content in Express.

You will learn the following in this chapter:

- The basics of HTTP response format
- How to set HTTP status code in Express
- How to set HTTP headers in Express
- How to serve different kinds of content in Express

A primer on HTTP response

To understand the types of response Express is capable of generating and have a better control over them, it is important that you have some technical understanding about the underlying HTTP protocol's response format. So, let's go over it real quick and cover the basics.

HTTP response is a small part of the much bigger HTTP protocol. As a web developer it is an added advantage if you have a good understanding of the protocol you are working with. You can read about the HTTP protocol in detail at http://www.w3.org/Protocols/.

For More Information:
www.packtpub.com/express-web-application-development/book
The data sent by an HTTP server in response to a request is called an **HTTP response message**. It is composed of a status code, headers, and optional associated data, which is technically referred to as the body of the message.

**HTTP status codes**

You might be familiar with 404 and 500 errors already. These error names are derived from the HTTP status code used to convey the errors.

404 and 500 are examples from the number of HTTP status codes that can be sent by the server to the client. Although there are a number of HTTP status codes, the reason you don't get to see them all is, because these codes are targeted at the user agent and the exchange takes place in the background.

An HTTP user agent is any software that a user makes use of to make requests to a web server. In most cases, the user agent happens to be a web browser, so we will be using the terms "user agent" and "browser" interchangeably in this book.

All responses from an HTTP server come with an associated status code. The most common among them is 200—the code for a successful request. Here is an example of a 200 HTTP status code:
Although not visible to the regular user, the HTTP status code sent by the server can be seen in the Network tab of most browser debugging tools.

Here is an example of a 404 status code sent by the server when requested for a non-existent resource:

As mentioned a while ago, HTTP status codes are not limited to just 200, 404, and 500; there are many more used for conveying many different kinds of messages.

Since we are working with Express, which is an HTTP server, it makes sense to be the aware about all the HTTP status codes, even if we don't get into the finer details about them.
Following is a list of all the HTTP status codes for your information and general knowledge:

It is beyond the scope of this book to get into all the details of HTTP status code, but you can find them all at http://www.
w3.org/Protocols/rfc2616/rfc2616-sec10.html.

**1xx**
The 1xx series of status codes is classified as **Informational**, and is used for conveying provisional response from the server.

The available codes in this series are: 100, 101, and 102.

**2xx**
The 2xx series of status codes is classified as **Success**, and is used for conveying a successful request for a resource on the server.

The available codes in this series are: 200, 201, 202, 303, 204, 205, 206, 207, 208, 250, and 226.

**3xx**
The 3xx series of status codes is classified as **Redirection**, and is used for information by the user agent about taking additional action to retrieve the requested resource.

The available codes in this series are: 300, 301, 302, 303, 304, 305, 306, 307, and 308.

**4xx**
The 4xx series of status codes is classified as **Client Error**, and is used for informing the user agent of its erroneous requests to the server.


For More Information:
www.packtpub.com/express-web-application-development/book
5xx

The 5xx series of status codes is classified as Server Error, and is used for informing the user agent that the server has encountered an error because of which the request was not fulfilled.

The available codes in this series are: 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 551, 598, and 599.

HTTP response headers

HTTP response headers (often referred to as just headers) are key-value pairs sent after the HTTP status code in a HTTP message. These headers are used for conveying various important pieces of information from the server to the user agent.

The following is an example of headers sent by a web server:

- X-Powered-By: Express
- Accept-Ranges: bytes
- ETag: "819254-1356021445000"
- Date: Mon, 11 Mar 2013 21:19:05 GMT
- Cache-Control: public, max-age=0
- Content-Type: image/gif
- Content-Length: 819254
- Connection: keep-alive

HTTP headers, like HTTP status codes, are targeted at the user agent, rather than the user, so they are also not visible to a regular user. However, they can be seen using network traffic analyzers and browser debugging tools such as Firebug and Chrome Developer Tool.

The HTTP protocol specifies a standard set of headers and possible values, which can be set in a message; however, there is no technical restriction on the actual implementation. Having said that, it is recommended to follow the standards to ensure the app works in a predictable manner.

Media types

Media type describes the kind of data that is being transferred over the Internet protocol; in our case it would be the HTTP protocol.

You might already be familiar with terms such as text/html, multipart/form-data, text/plain, and so on, those are examples of description of media types.

For More Information:
www.packtpub.com/express-web-application-development/book
Media Type is also commonly referred to as MIME Type or Content Type. You can read more about them at http://www.iana.org/assignments/media-types.

Whenever an HTTP server sends a response, it also specifies what kind of data it is sending via the Content-Type header, which is shown in the following screenshot:

The Content-Type header can have an optional parameter, which specifies the encoding for the data being transferred. On the Web, this parameter is most commonly applicable to string data, such as plain text, HTML, and JSON:

UTF-8 is the most popular encoding format on the Web, and is the default in JavaScript, Node, and Express.

For More Information:
www.packtpub.com/express-web-application-development/book
HTTP response in Express

Now that we have covered the basics of HTTP response in general, let’s find out if it works in Express.

By now, we already know how a simple HTTP response can be generated in Express—by setting up a route and a handler for it:

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

Let’s examine the response for this response.

Express can send an HTTP response using one of its response methods: `res.send()`, `res.json()`, `resjsonp()`, `res.sendFile()`, `res.download()`, `res.render()`, or `res.redirect()`. If none of them is called, the request will be left hanging till the connection times out.

If more than one response methods are specified in a route handler, only the first method will take effect, the rest will generate non-fatal, run-time errors.

Start the app, load the homepage, and look at the response headers using a browser-debugging tool:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 7
Date: Sat, 09 Mar 2013 15:55:24 GMT
Connection: keep-alive
```

The HTTP status is set to `200 OK`, the content type is set to `text.html; charset=utf-8`, and a bunch of other HTTP headers have been set. Express does all of these for you, automatically for successful requests.

It is not limited to successful results only, Express also handles 404 errors for you. Try making a request to a non-existent URL on your local machine, for example, `http://localhost:3000/foo`, and examine the response headers:

```
HTTP/1.1 404 Not Found
X-Powered-By: Express
Content-Type: text/plain
```

For More Information:

www.packtpub.com/express-web-application-development/book
In fact, Express does all that is expected from any decent web server. Naturally, it handles 500 errors too. Modify the home page handler to cause a runtime error:

```javascript
app.get('/', function(req, res) {
    // Call an undefined function
    error();
});
```

Restart the app, load the homepage, and examine the response headers again:

```
HTTP/1.1 500 Internal Server Error
X-Powered-By: Express
Content-Type: text/plain
Content-Length: 938
Date: Sat, 09 Mar 2013 18:06:13 GMT
Connection: keep-alive
```

So Express takes care of setting the right HTTP status code and headers for the requests made to it, and we really don't have to do anything much at all. It is all well and good, but being curious hackers, we start to wonder if there are ways to customize the HTTP status code and the headers. Can we?

## Setting the HTTP status code

Setting the HTTP status code is as easy as passing a number to the `res.status()` method. Let's look at some examples of to find out how `res.status()` works.

```
res.status() alone is not enough to handle the response. It needs to be used in conjunction with one of the Express HTTP response methods, such as `res.send()`, `res.render()`, and so on, or else the request will be left hanging.
```

In the following example, we send a status code of 404 even though the home page route actually exists. If we hadn't specified 404, Express would have sent a status code of 200:

```javascript
app.get('/', function(req, res) {
    // Set the status
    res.status(404);
    // Specify the body
    res.send('forced 404');
});
```
And in this example, we send a 500 status code:

```javascript
app.get('/', function(req, res) {
    res.status(500);
    res.send('forced 500');
});
```

`res.status()` is a chainable method, meaning we can do things like the following:

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

Among the HTTP response methods, `res.send()`, `res.json()`, and `res.jsonp()` are capable of specifying the HTTP status code themselves, without the help of `res.status()`. When a status code is not specified, a default of 200 is assigned.

Here are some examples showing how it works using `res.send()`:

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

When a number alone is passed to `res.send()`, it is assumed to be the intended status code. The server will just send the status code and the basic headers, with no body:

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

It is your responsibility to pass the proper HTTP status code to the response methods; Express won’t check the validity of the number. Whatever you pass will be sent to the user agent.

And of course, you can set the status code and the body at the same time, like so:

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

For `res.render()`, `res.sendfile()`, and `res.download()`, you will need to specify the status code using the `res.status()` method, or else it will default to 200:

```javascript
app.get('/', function(req, res) {
    res.status(404);
});
```
Response From the Server

```
res.status(404);
res.render('index', {title: 'Express'});
```
If the process of individually setting the headers seems tedious to you, you can use the alternative approach of passing an object to `res.set()` instead. The key-value pair in the object will be assigned as the header and its value in the HTTP response message:

```javascript
app.get('/', function(req, res) {
    res.set({'
        'Content-Type': 'text/plain; charset=utf-8',
        'X-Secret-Message': 'not really secret',
        'X-Test': 'OK'
    });
    res.send('welcome');
});
```

Very related to setting HTTP headers, Express provides a `res.charset` property, which can be used to set the value of the optional `Content-Type` header. This property is best used when you just want to change the charset of the default `Content-Type` of `text/html`.

The following code will set the `Content-Type` header to `text/html; charset=utf-8`:

```javascript
app.get('/', function(req, res) {
    res.charset = 'utf-8';
    res.send('welcome');
});
```

Now that we have covered setting the HTTP status code and headers, it is about time that we started learning about sending stuff to the users that they can actually see and interact with.

For More Information:

www.packtpub.com/express-web-application-development/book
Sending data

The component of an HTTP response message, which users can generally see and interact with, is called the body of the message. The body can come in many different forms—as HTML, plain text, images, binary files, CSS files and so on—and the Content-Type header is exclusively used to convey to the user agent what sort of data it is dealing with.

Let's find out how different kinds of data can be served from an Express application.

Plain text

One may wonder if we actually need to dedicate a section on how to send plain text from Express. We will soon find out whether it was worth it or not.

Let's create a very simple route handler for the home page route. Our intention is to see the HTML tags as is in the browser:

```javascript
app.get('/', function(req, res) {
  res.send('<h1>welcome</h1>');
});
```

Start the app, load the homepage, and analyze the output:

![Browser rendering the output as HTML](image)

The browser actually rendered the output as HTML. Maybe you expected it, or maybe you did not; but we need to get to the root of this behavior.

The browser interpreted the output as HTML and rendered it accordingly, because the default value of the Content-Type header in Express is text/html.

However, we wanted the browser to treat the output as plain text and not bother rendering it as HTML.

For More Information:

Is there a way to enforce that behavior? Yes, there is a way. Using our newfound knowledge of HTTP headers and Express' `res.set()` method, we can set the `Content-Type` header to `text/plain`.

Update the code accordingly to confirm our assumption:

```javascript
app.get('/', function(req, res) {
  res.set('Content-Type', 'text/plain');
  res.send('<h1>welcome</h1>');
});
```

Restart the app and reload the home page:

This time the content was really treated as plain text, like how we wanted. When the `Content-Type` is set to `text/plain`, the browser will render the body as plain text—this is the case even for binary files.

Although the exercise was about sending plain text data, we had a very good demonstration about the power of HTTP headers, especially the `Content-Type` header.

**HTML**

Being an HTTP server, sending the content as HTML is the default behavior of Express. Anything you send via `res.send()` or `res.render()` is sent as HTML by setting the `Content-Type` header to `text/html`.

For More Information:
As seen in a previous example, the following response body will be interpreted and rendered as HTML by the browser:

```
app.get('/', function(req, res) {
    res.send('<h1>welcome</h1>');
});
```

And in this example, Express will render a view using the Jade template engine, and generate a corresponding HTML page:

```
app.use(express.static('./public'));
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');

app.get('/', function(req, res) {
    res.render('index', {title:'Express'});
});
```

If you look at the source code of the HTML code generated by Jade, you will find that the whole HTML is in a single line, as shown in the following screenshot:

![HTML code generated by Jade](image)

For whatever reason, you may sometimes want the HTML to be pretty-printed—show tags in separate lines with appropriate indentation to show hierarchy. To enable that, just set the value of `app.locals.pretty` in your app to `true`.

The previous code can be re-written as follows to prettify the generated HTML:

```
app.use(express.static('./public'));
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
```
// HTML should be prettified
app.locals.pretty = true;

app.get('/', function(req, res) {
    res.render('index', {title:'Express'});
});

Restart the server, reload the home page, and look at the source code now:

```
<html>
<head>
    <title>Express</title>
    <script src="javascripts/main.js"></script>
    
    <link rel="stylesheet" href="stylesheets/style.css"/>
</head>
<body>
    <div id="content"><img src="images/logo.png"/>
        <p>WELCOME</p>
        <p></p>
    </div>
</body>
</html>
```

The generated HTML will now be pretty-printed. However, in a production environment it is best not to prettify HTML to save some processing power and reduce the download size of the HTML page.

**JSON**

Express provides the `res.json()` method for serving JSON content. You just have to pass an object to it, and it will take care of setting the right headers and formatting the JSON string according to the JSON specifications.

Create this route for the home page:

```javascript
app.get('/', function(req, res) {
    res.json({message: 'welcome'});
});
```

For More Information:

www.packtpub.com/express-web-application-development/book
Start the app, load the home page, and examine the HTTP response headers:

```

{  "message": "welcome"
}
```

res.json() has successfully transformed the JavaScript object to a valid JSON string and set the appropriate HTTP headers for the message.

Like other response methods, res.json() sets a default of 200 when no status code is explicitly set. You can customize the status code by passing a number as the first parameter of res.json(), followed by the object to be sent:

```
res.json(404, {error: 'not found'});
```

Unlike res.send(), if you pass just a number to res.json(), it will be interpreted as the intended JSON object, and the default status code of 200 will be sent instead of using it as the status code.

**JSONP**

**JSON with Padding (JSONP)** is a JavaScript technique to allow cross-domain scripts to execute callbacks from JSON requests made to an external domain. It is beyond the scope of this book to cover JSONP in detail, but you can read it up at [http://en.wikipedia.org/wiki/JSONP](http://en.wikipedia.org/wiki/JSONP).

A JSONP request comes with a GET request parameter, conventionally named **callback**, which is the callback function available at the website making the request, which will be executed by passing the JSON result from the external domain.

JSONP requests to Express are handled by the res.jsonp() method. This method works like res.json(), except it wraps the JSON result with the callback function specified in the request.

---

**For More Information:**

Let's define the route of the home page to respond with `res.jsonp()`:

```javascript
app.get('/', function(req, res) {
    res.jsonp({message: 'welcome'});
});
```

Start the server, load `http://localhost:3000/?callback=json_callback` in your browser, and examine the result:

Not only did `res.jsonp()` wrap the JSON result with the callback function, it also added a quick check for the existence of the callback on the client machine before executing the callback. Also, it set the `Content-Type` header to the appropriate `text/javascript` content type so that the browser interprets the result as JavaScript.

By default, `res.jsonp()` expects the name of the callback parameter to be named `callback`, but it can be renamed to anything you like using the `app.set()` method, as shown here:

```javascript
app.set('jsonp callback name', 'cb');
```

Now the callback name will be expected to be found in the `GET` parameter named `cb`. If the callback name is not found in the expected `GET` parameter, only the JSON object will be sent, without the callback padding.

### Serving static files

As we saw in [Chapter 2, Your First Express App](#), serving static files is very easy in Express—just set up a static directory using the static middleware and place the files there.
Create a directory named files in the app directory, keep the files in the directory, and add the following to the app file:

```javascript
// Use the static middleware to set up a static files directory
app.use(express.static('./files'));
```

Now you can access all the files in the directory from the root of the website. This is how static files for the app, such as CSS, JavaScript, and image files are served in Express.

If you have a file named logo.png in the files directory, you can access it at http://localhost:3000/logo.png. Any file or subdirectory you create in the files directory will also be correspondingly accessible from the app, for example, http://localhost:3000/new-logo.png, http://localhost:3000/icons/packt.png, and so on.

### Serving files programmatically

There is another category of files that can be served by a web server—those that are served dynamically—the requests to which you can apply programming logic.

Express provides two methods of handling such requests: `res.sendfile()` and `res.download()`. Let's examine them one after another.

> Note, it is `res.sendfile()`, not `res.sendFile()`.

Using `res.sendfile()`, you can send files to the browser in the same manner as how regular files are sent to it. The `Content-Type` header is automatically set based on the file extension, and depending on the file type and browser settings, the file may be shown in the browser, displayed by a plugin, prompted for download, and so on.

The following is a very simple example of using `res.sendfile()`:

```javascript
app.get('/file', function(req, res) {
  res.sendfile('./secret-file.png', function(err) {
    if (err) { console.log(err); }
    else { console.log('file sent'); }
  });
});
```

In this example, we send a private file from a private directory, to GET requests to the path/file on the server.

For More Information:
www.packtpub.com/express-web-application-development/book
In a web server context, public files and directories can be accessed via a URL, whereas private files and directories are those that are not exposed to the general public via a URL.

On loading `http://localhost:3000/file`, and examining the HTTP headers, we will find that no information about the actual name or location of the file was sent to the browser:

```javascript
app.get('/file.html', function(req, res) {
    console.log('HTML file is an image?');
    res.sendfile('./secret-file.png');
});
```

Considering the fact that routes names are very flexible and configurable in Express, you can do all sorts of useful or crazy things, when combined with `res.sendfile()`.

Here is an example that belongs to the crazy category:

```javascript
app.get('/file.html', function(req, res) {
    console.log('HTML file is an image?');
    res.sendfile('./secret-file.png');
});
```

For More Information:
www.packtpub.com/express-web-application-development/book
There are times when you want the user to actually download the file, and not let the browser try to render it. This can be achieved using the `res.download()` method.

```javascript
app.get('/download', function(req, res) {
  res.download('./secret-file.png', 'open-secret.png', function(err) {
    if (err) { console.log(err); } else { console.log('file downloaded'); }
  });
});
```

If you examine the HTTP headers for this response, you will find that the `Content-Disposition` header has been set to attachment, because of which the file is being prompted for download or being downloaded:

If a filename is not specified for the download to `res.download()`, the original name of the file will be used.

**Serving error pages**

Displaying an error page can be as simple as sending just an error status code with no body, or rendering an elaborate 404 or 500 error page.

For More Information:

www.packtpub.com/express-web-application-development/book
The simplest way to display an error page is to just send the HTTP error code. In this case, the browser will "know" about the error, but the user will see just a blank screen.

```javascript
// 404 error
res.send(404);
```

You can elaborate this a little bit more by adding a body that will be displayed in the browser. Now even the user will be aware about the error:

```javascript
// 404 with additional message body
res.send(404, 'File not Found');
```

Using `res.render()`, you can have beautifully customized error pages for your website, if you want to.

In theory, using `res.send()` and `res.render()` to serve error pages sounds very straightforward and easy, however, in reality handling 404 and 500 errors is not that obvious to most beginners. Let's find out how we can catch these errors and send the appropriate responses.

We will be using views in the upcoming examples, so make sure you have set the `views` directory in the `app.js` file, or else the examples will fail to work. Refer Chapter 2, Your First Express App, for setting up views for your app.

The `router` middleware comes with a default 404 error handler, but its output may not be what you would want for your app. Let's find out how to create a custom 404 error handler.

A 404 error handler is technically a generic route handler that handles a request that all other middleware before it has failed to handle. It is implemented by adding a custom middleware at the end of the Express middleware stack.

Add the following middleware code after the `router` middleware:

```javascript
app.use(function(req, res) {
    res.status(400);
    res.render('404.jade', {
        title: '404',
        message: 'File Not Found'
    });
});
```
When the in-built 404 error handler detects that there is a route handler even beyond it, it will pass on the request to the next handler, which would be our custom 404 error handler.

In the views directory, create a file named 404.jade with the following content:

```jade
doctype 5
html
  head
    title #{title}
  body
    h1 #{title}
    p #{message}
```

Restart the server and load a non-existent URL to see the 404 error page:

![404 Error Page](Image)

There you have it, your custom 404 error page! Feel free to modify and customize 404.jade to your maximum satisfaction.

Express also comes with a default 500 error handler that will pass on the control to the next error handler, if there is one beyond it.

The 500 error is handled by adding a middleware with an arity of four. Since we want to override the default 500 error handler provided by the router middleware, we would need to add our handler after the router middleware.

Add the following middleware after the router middleware:

```javascript
app.use(function(error, req, res, next) {
  res.status(500);
  res.render('500.jade',
```

For More Information:

www.packtpub.com/express-web-application-development/book
Now create the corresponding 500.jade view file in the views directory:

```jade
doctype 5
html
  head
    title #{title}
  body
    h1 #{title}
    p #{error}
```

To intentionally cause a 500 error, create a route with a callback that tries to execute an undefined function:

```javascript
app.get('/error', function(req, res) {
  // Call an undefined function
  error();
});
```

Restart the app, load http://localhost:3000/error in your browser to see the 500 error page:

![500 Error Page](image)

There you go, your own custom 500 error page!

For More Information:
www.packtpub.com/express-web-application-development/book


Response From the Server

Content negotiation

Content negotiation is the mechanism of specifying the data types a user agent is capable of consuming and prefers, and the server fulfilling the request when it can, and informing when it cannot.

User agents send their preferred content type for a resource using the `Accept` HTTP request header.

Express supports content negotiation using the `res.format()` method. This is a useful feature if you want to send different types of content based on the capability of the user agent.

`res.format()` accepts an object whose keys are the canonical content type name (text/plain, text/html, and so on), and whose values are functions that will be used as the handler for the route, for the matching content type.

Let's implement content negotiation in the home page route handler to find out how it works:

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

The server will respond with the appropriate data type based on the `Accept` header. This fact can be verified by sending an `Accept` header of text/sgml, application/json:

For More Information:
www.packtpub.com/express-web-application-development/book
Similarly, you will get the corresponding content type if you set the `Accept` header to `text/plain` or `text/html`.

If a user agent does not support any of the specified formats in the handler, the server will return a status of **406 Not Acceptable**.

The previous code can be re-written in a less verbose manner by using just the subtype of the content type as the key:

```javascript
res.format({
  text: function() {
    res.send('welcome');
  },

  html: function() {
    res.send('<b>welcome</b>');
  },

  json: function() {
    res.json({ message: 'welcome' });
  },

  default: function() {
    res.send(406, 'Not Acceptable');
  }
});
```

---

For More Information:

The default handler is optional. When not defined, an unsuccessful content negotiation will be handled by Express' built-in implementation of 406.

Redirecting a request

Sometimes you may want to redirect the request to a different URL, instead of responding with data. This is made possible in Express using the `res.redirect()` method. This method takes an optional redirection code that defaults to 302, and the URL to redirect to. The `URL` parameter can be an absolute URL or relative to the current URL.

The following are some examples of redirecting requests from an Express app:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>res.redirect('/notice');</code></td>
<td>302 redirection to /notice relative to the requested URL</td>
</tr>
<tr>
<td><code>res.redirect(301, '/help-docs');</code></td>
<td>301 redirection to /help-docs relative to the requested URL</td>
</tr>
<tr>
<td><code>res.redirect('http://nodejs.org/api/');</code></td>
<td>301 redirection to an absolute URL</td>
</tr>
<tr>
<td><code>res.redirect('../images');</code></td>
<td>302 redirection to /notice relative to the requested URL</td>
</tr>
</tbody>
</table>

Summary

We now know that a lot more happens in the background when we load something in the browser. We learned to customize the HTTP response object in Express to control the outcome of the request and serve different content types from our app.

We were introduced to Jade in *Chapter 2, Your First Express App*, and it has been a constant presence in all the chapters till now. So far we know that it is a templating engine and works great for generating HTML from our app. There is much more to Jade than what we saw in the examples, in the next chapter we will learn about it in greater detail.
Where to buy this book
You can buy Express Web Application Development from the Packt Publishing website:

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.