UNIT- 3

Syllabus:

Implementing HTTP Services in Node.js:

- Processing URLs,
- Understanding Request, Response, and Server Objects,
- Implementing HTTP Clients and Servers in Node.js,
- Implementing HTTPS Servers and Clients

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.

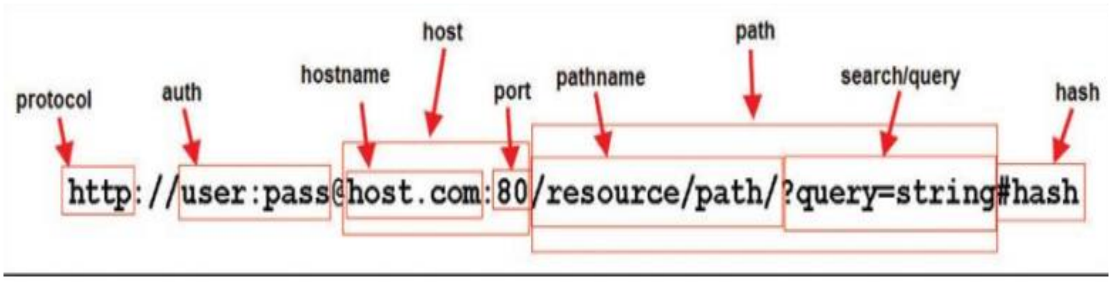

protocol
auth
hostname
host
port
pathname
path
search/query
hash
http://user:pass@host.com:80/resource/path/?query=string#hash

Figure 7.1 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 `url.parse()` method takes a URL string, parses it, and returns a URL object.
- A `TypeError` is thrown if `urlString` is not a string.

- 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.
- parseQueryString <boolean> If true, the query property will always be set to an object returned by the querystring module's parse() method. If false, the query property on the returned URL object will be an unparsed, undecoded string. **Default:** false.

A `URIError` is thrown if the `auth` property is present but cannot be decoded.

`url.parse()` uses a lenient, non-standard algorithm for parsing URL strings. It is prone to security issues such as host name spoofing and incorrect handling of usernames and passwords. Do not use with untrusted input. CVEs are not issued for `url.parse()` vulnerabilities. Use the WHATWG URL API instead.

- 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#hash';
  var urlObj = url.parse(urlStr, true, false);
  urlString = url.format(urlObj);
  ```

urlObject.auth

The `auth` property is the username and password portion of the URL, also referred to as *userinfo*. This string subset follows the `protocol` and double slashes (if present) and precedes the `host` component, delimited by `@`. The string is either the username, or it is the username and password separated by `:`.

For example: `'user:pass'`.

`urlObject.hash`

The `hash` property is the fragment identifier portion of the URL including the leading `#` character.

For example: `'#hash'`.

`urlObject.host`

The `host` property is the full lower-cased host portion of the URL, including the `port` if specified.

For example: `'sub.example.com:8080'`.

`urlObject.hostname`

The `hostname` property is the lower-cased host name portion of the `host` component *without* the `port` included.

For example: `'sub.example.com'`.

`urlObject.href`

The `href` property is the full URL string that was parsed with both the `protocol` and `host` components converted to lower-case.

For example: `'http://user:pass@sub.example.com:8080/p/a/t/h?query=string#hash'`.

`urlObject.path`

The `path` property is a concatenation of the `pathname` and `search` components.

For example: `'/p/a/t/h?query=string'`.

No decoding of the `path` is performed.

`urlObject.pathname`

The `pathname` property consists of the entire path section of the URL. This is everything following the `host` (including the `port`) and before the start of the `query` or `hash` components, delimited by either the ASCII question mark (`?`) or hash (`#`) characters.

For example: `'/p/a/t/h'`.

No decoding of the path string is performed.

`urlObject.port`

The `port` property is the numeric port portion of the `host` component.

For example: `'8080'`.

`urlObject.protocol`

The `protocol` property identifies the URL's lower-cased protocol scheme.

For example: `'http:'`.

`urlObject.query`

The `query` property is either the query string without the leading ASCII question mark (`?`), or an object returned by the `querystring` module's `parse()` method. Whether the `query` property is a string or object is determined by the `parseQueryString` argument passed to `url.parse()`.

For example: `'query=string'` or `{'query': 'string'}`.

If returned as a string, no decoding of the query string is performed. If returned as an object, both keys and values are decoded.

`urlObject.search`

The `search` property consists of the entire "query string" portion of the URL, including the leading ASCII question mark (`?`) character.

For example: `'?query=string'`.

No decoding of the query string is performed.

`urlObject.slashes`

The `slashes` property is a `boolean` with a value of `true` if two ASCII forward-slash characters (`/`) are required following the colon in the `protocol`.

RESOLVING URL COMPONENTS:

- Another useful feature of the url module is the ability to resolve URL components in the same manner as a browser would.
- This allows you to manipulate the URL strings on the server side to make adjustments in the URL.
- For example, you might want to change the URL location before processing the request because a resource has moved or changed parameters.
- 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 of the previous code snippet is shown below. Notice that only the resource path and beyond are altered in the resolved URL location: http://user:pass@host.com:80/another/path?querynew

PROCESSING QUERY STRINGS AND FORM PARAMETERS:

- HTTP requests often include query strings in the URL or parameter data in the body for form submissions.
- The query string and form parameters are just basic key-value pairs. To actually consume these values in your Node.js webserver you need 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 following shows an example of using parse() to parse a query string:

  ```
  var qstring = require('querystring');
  var params = qstring.parse("name=Brad&color=red&color=blue");
  ```

  The params object created would be: {name: 'Brad', color: ['red', 'blue']}
- You can also go back the other direction and convert an object to a query string using the stringify() function shown here: querystring.stringify(obj, [sep], [eq]).

Understanding Request, Response, and Server Objects

In Node.js, HTTP services use request, response, and server objects to handle incoming requests and generate outgoing responses. Here is a brief overview of each object:

- **Request Object**: The request object represents the incoming HTTP request from the client. It contains information about the request, such as the request method (GET, POST, etc.), headers, URL, and query parameters. Developers can access this information using various properties and methods of the request object.
- **Response Object**: The response object represents the outgoing HTTP response from the server. It is used to send data back to the client, such as HTML, JSON, or files. Developers can use various methods and properties of the response object to set the response status code, headers, and body.
- **Server Object**: The server object represents the HTTP server instance created by Node.js. It listens for incoming requests and dispatches them to the appropriate request handlers. Developers can use methods and properties of the server object to configure the server, such as setting the port number, registering request handlers, and starting or stopping the server.
- Together, these objects provide a powerful framework for building HTTP services in Node.js. Developers can use them to handle incoming requests, generate responses, and manage the server itself.

The http.ClientRequest Object

- The ClientRequest object is created internally when you call http.request() when building the HTTP client.
- This object represents the request while it is in progress to the server. You use the ClientRequest object to initiate, monitor, and handle the response from the server.

- The ClientRequest 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 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 below Table lists the properties that you can specify. 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 only parameter to the callback is an IncomingMessage object that will be the response from the server.

The following code shows the basics of implementing the ClientRequest object:

```
const http = require('http');

const server = http.createServer((req, res) => {

  console.log('Request method:', req.method);

  console.log('Request URL:', req.url);

  console.log('Query parameters:', req.url.query);

  res.end('Hello World!');

});

server.listen(3000, () => {

  console.log('Server is listening on port 3000');

});
```

In this example, we create an HTTP server that listens on port 3000. When a request comes in, we log the request method, URL, and query parameters to the console. Finally, we send back a simple response of "Hello World!" using the response object's end() method.

Options that can be specified when creating a ClientRequest

In Node.js, the http.request() and https.request() methods are used to create a ClientRequest object to make HTTP/HTTPS requests to a server. When creating a ClientRequest object, you can specify the following options:

- **protocol:** The protocol to use. This can be either 'http:' or 'https:'.
- **host:** The hostname or IP address of the server.
- **hostname:** Same as host, but preferred to be used instead of host.
- **port:** The port number to use. Defaults to 80 for HTTP and 443 for HTTPS.
- **path:** The path of the resource to request. This can include a query string.
- **method:** The HTTP method to use. Defaults to 'GET'.

- **headers:** An object containing request headers.
- **auth:** The authentication credentials to use in the format 'username:password'.
- **agent:** The HTTP/HTTPS agent to use for the request.
- **timeout:** The maximum time to wait for a response before aborting the request.
- **family:** The IP address family to use. This can be either '4' or '6'.
- **localAddress:** The local IP address to use when making the request.
- **Socketpath:** Unix domain Socket(use one of host:port or socketpath).

Events available on ClientRequest objects

In Node.js, the ClientRequest object is used to make HTTP/HTTPS requests to a server. The following events are available on ClientRequest objects:

- **response:** This event is emitted when the server responds to the request. The response object is passed as an argument to the event handler.
- **error:** This event is emitted if an error occurs while making the request. The error object is passed as an argument to the event handler.
- **abort:** This event is emitted when the request is aborted. This can happen if the abort() method is called on the ClientRequest object.
- **socket:** This event is emitted when a socket is assigned to the request.
- **timeout:** This event is emitted if the request times out before a response is received from the server.

To handle these events, you can use the on method on the ClientRequest object, like this:

```
const req = http.request(options, (res) => {

  // handle response

});

req.on('error', (err) => {

  // handle error

});

req.on('abort', () => {

  // handle abort

});

req.on('socket', (socket) => {

  // handle socket assignment

});

req.on('timeout', () => {

// handle timeout

});
```

Note that these events are also available on the http.ClientResponse object, which is the response object passed to the response event handler.

Methods available on ClientRequest objects

- **write(chunk[, encoding][, callback]):** This method writes chunk to the request body. If encoding is provided, it is used to encode chunk. If a callback function is provided, it is called when the chunk has been written**.**
- **abort():** This method aborts the request.
- **end([data][, encoding][, callback]):** This method sends the request to the server. If data is provided, it is sent as the request body. If a callback function is provided, it is called when the response is received.
- **setNoDelay([noDelay]):** This method sets the noDelay option on the underlying socket. If noDelay is true, data is sent immediately. If noDelay is false, data is buffered until the buffer is full or the socket is closed.
- **setTimeout(timeout[, callback]):** This method sets a timeout for the request. If the server does not respond within the specified timeout, the timeout event is emitted. If a callback function is provided, it is called when the timeout occurs.
- **setSocketKeepAlive([enable][, initialDelay]):** This method sets the keepAlive and initialDelay options on the underlying socket. If enable is true, the socket is kept alive. If initialDelay is provided, it sets the initial delay for sending keepalive packets.

These methods can be used to configure and send HTTP/HTTPS requests to a server using the ClientRequest object in Node.js.

```
const https = require('https');

const options = {

  hostname: 'www.example.com',

  port: 443,

  path: '/',

  method: 'GET',

};

const req = https.request(options, (res) => {

  console.log(`statusCode: ${res.statusCode}`);

  res.on('data', (d) => {

    process.stdout.write(d);

  });

});

req.on('error', (error) => {

  console.error(error);
});
// Set a timeout for the request
req.setTimeout(5000, () => {
  console.log('Request timed out');
  req.abort();
});
// Write data to the request body
req.write('Hello, world!');
req.end();
```

- In this example, we are making an HTTPS GET request to www.example.com. We set a timeout of 5 seconds for the request using the setTimeout method, and we write data to the request body using the write method. Finally, we end the request using the end method. We also handle any errors that occur during the request using the error event.

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.
- When handling the client request, you use the properties, events, and methods of the ServerResponse object to build and send headers, write data, and send the response. The belowTable lists the event and properties available on the ServerResponse object. The below Table lists the methods available on the ServerResponse object.

The following code shows the basics of implementing the ServerResponse object:

```
const http = require('http');
const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('<h1>Hello World!</h1>');
  res.end();
});
server.listen(3000, () => {
  console.log('Server is listening on port 3000');
});
```

In this example, we create an HTTP server that listens on port 3000. When a request comes in, we use the response object's writeHead() method to set the response status code to 200 and the Content-Type header to text/html. We then use the write() method to send an HTML response, and finally, we call the end() method to finish the response.

Events available on ServerResponse objects

In Node.js, the ServerResponse object is an instance of the http.ServerResponse class, which represents the HTTP response that an HTTP server sends back to the client.

- The ServerResponse object emits several events during the lifetime of the HTTP response. Here are some of the commonly used events available on ServerResponse objects in Node.js:
- **'close':** Emitted when the response has been closed. This event is triggered once the response has been sent to the client, and the connection has been closed.
- **'finish':** Emitted when the response has been completely written to the client. This event is triggered once the response has been sent to the client, and all data has been written.
- **'headersSent':** Emitted when the response headers have been sent to the client. This event is triggered once the headers have been sent, and the client may start receiving data.
- **'error':** Emitted when an error occurs while sending the response. This event is triggered if an error occurs while sending data to the client.

You can listen to these events using the .on() method, like this:

```
response.on('event', function() {
  // handle the event
});
```

Where response is the ServerResponse object, and 'event' is the name of the event you want to listen to.

Methods available on ServerResponse objects

In Node.js, the ServerResponse object is an instance of the http.ServerResponse class, which represents the HTTP response that an HTTP server sends back to the client.

The ServerResponse object provides several methods to manipulate the HTTP response. Here are some of the commonly used methods available on ServerResponse objects in Node.js:

- **response.write(chunk[, encoding][, callback]):** This method writes data to the response stream. The chunk parameter can be a string or a buffer, and the encoding parameter specifies the encoding of the chunk. The optional callback parameter is called when the data has been flushed to the underlying system.
- **response.end([data][, encoding][, callback]):** This method signals that the response has been sent completely. The optional data parameter can be used to write the last chunk of data to the stream. The encoding parameter specifies the encoding of the data. The optional callback parameter is called when the data has been flushed to the underlying system.

- **response.setHeader(name, value):** This method sets a single header on the response. The name parameter specifies the header name, and the value parameter specifies the header value.
- **response.writeHead(statusCode[, statusMessage][, headers]):** This method sends the response headers to the client. The statusCode parameter specifies the HTTP status code, the optional statusMessage parameter specifies the HTTP status message, and the optional headers parameter specifies additional headers to send.
- **response.status(statusCode):** This method sets the HTTP status code of the response.
- **response.sendDate = true/false:** This property sets whether or not the server should include the Date header in the response.

  ```
  const http = require('http');
  const server = http.createServer((req, res) => {
    // Set the response headers
    res.setHeader('Content-Type', 'text/html');
    res.setHeader('X-Powered-By', 'Node.js');
    // Write some data to the response
    res.write('<html><head><title>My Page</title></head><body>');
    res.write('<h1>Welcome to my page!</h1>');
    res.write('<p>This is a sample page served by Node.js.</p>');
    res.write('</body></html>');
    // End the response and send it to the client
    res.end();
  });
  server.listen(3000, () => {
    console.log('Server listening on port 3000');
  });
  ```
- In this example, we create an HTTP server using the http.createServer() method, and set up a request handler that sets some response headers using the res.setHeader() method. We then use the res.write() method to write some data to the response stream, and finally use the res.end() method to end the response and send it to the client.

The http.Server Object

- The Node.js HTTP Server object provides the fundamental framework to implement HTTP servers. It provides an underlying socket that listens on a port and handles receiving requests and then sends responses out to client connections.
- While the server is listening, the Node.js application will not end. The Server object implements EventEmitter and emits the events listed in the below Table. As you implement an HTTP server, you need to handle at least some or all of these events.
- For example, at a minimum you need an event handler to handle the request event that is triggered when a client request is received.

The following example shows how to use the server object to start and stop the server:

```
const http = require('http');

const server = http.createServer((req, res) => {
```

```
  res.end('Hello World!');
});
server.listen(3000, () => {
  console.log('Server is listening on port 3000');
});
// Stop the server after 10 seconds
setTimeout(() => {
  console.log('Stopping server...');
  server.close(() => {
    console.log('Server has been stopped');
  });
}, 10000);
```

In this example, we create an HTTP server that listens on port 3000. We then use the setTimeout() function to stop the server after 10 seconds by calling the server object's close() method. When the server has been stopped, we log a message to the console.

Events that can be triggered by Server objects

In Node.js, the Server object is an instance of the http.Server class, which represents an HTTP server that listens for incoming requests.

- The Server object provides several events that can be triggered during its lifetime. Here are some of the commonly used events that can be triggered by Server objects in Node.js:
- **'request':** Emitted when a new HTTP request is received by the server. The event handler is passed two arguments: an IncomingMessage object representing the request, and a ServerResponse object representing the response.
- **'connection':** Emitted when a new TCP connection is established. The event handler is passed a net.Socket object representing the connection.
- **'listening':** Emitted when the server starts listening for incoming requests. The event handler is passed no arguments.
- **'close':** Emitted when the server is closed using the server.close() method. The event handler is passed no arguments.
- **'error':** Emitted when an error occurs on the server. The event handler is passed an Error object representing the error.
- **'upgrade':** Emitted when a client requests an upgrade to a different protocol, such as WebSocket. The event handler is passed an IncomingMessage object representing the upgrade request, a net.Socket object representing the connection, and a Buffer object containing the first packet of the upgraded protocol.

To start the HTTP server, you need to first create a Server object using the createServer() method shown below. This method returns the Server object. The optional requestListener parameter is a callback that is executed when the request event is triggered.

- The callback should accept two parameters. The first is an IncomingMessage object representing the client request, and the second is a ServerResponse object you use to formulate and send the response:

  http.createServer([requestListener])
- Once you have created the Server object, you can begin listening on it by calling the listen() method on the Server object:

  listen(port, [hostname], [backlog], [callback])

- The first method listen(port, [hostname], [backlog], [callback]) is the one that you will most likely use. The following list describes each of the parameters:

  ■ **port:** Specifies the port to listen on.

  ■ **hostname:** Specifies when the hostname will accept connections, and if omitted, the server will accept connections directed to any IPv4 address (INADDR_ANY).

  ■ **backlog:** Specifies the maximum number of pending connections that are allowed to be queued. This defaults to 511.

  ■ **callback**: Specifies the callback handler to execute once the server has begun listening on the specified port.

- The following code shows an example of starting an HTTP server and listening on port 8080.

  Notice the request callback handler:

  ```
  var http = require('http');
  http.createServer(function (req, res) {
  <<handle the request and response here>>
  }).listen(8080);
  ```

- Two other methods can be used to listen for connections through the file system. The first accepts a path to a file to listen on, and the second accepts an already open file descriptor handle:

  listen(path, [callback])

  listen(handle, [callback])

- To stop the HTTP server from listening once it has started, use the following close() method:

  close([callback]).

Implementing HTTP Clients and Servers in Node.js

What is a Module in Node.js?

- Consider modules to be the same as JavaScript libraries.
- They are a set of functions you want to include in your application.

Built-in Modules:

- Node.js has a set of built-in modules which you can use without any further installation.

Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP). To include the HTTP module, use the require() method:

var http = require('http');

- The HTTP module can create an HTTP server that listens to server ports and gives a response back to the client. Use the createServer() method to create an HTTP server:

```
var http = require('http');
//create a server object:
http.createServer(function (req, res) {
  res.write('Hello World!'); //write a response to the client
  res.end(); //end the response
}).listen(8080); //the server object listens on port 8080
```

The function passed into the http.createServer() method, will be executed when someone tries to access the computer on port 8080.

Create Your Own Modules:

- You can create your own modules, and easily include them in your applications.
- The following example creates a module that returns a date and time object:

Example: Create a module that returns the current date and time:

```
exports.myDateTime = function () {
  return Date();
};
```

- Use the exports keyword to make properties and methods available outside the module file. Save the code above in a file called "myfirstmodule.js".

Include Your Own Module:

- Now you can include and use the module in any of your Node.js files.

Example: Use the module "myfirstmodule" in a Node.js file:

```
var http = require('http');
var dt = require('./myfirstmodule');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write("The date and time are currently: " + dt.myDateTime());
  res.end();
}).listen(8080);
```

The Built-in HTTP Module:

- Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).
- To include the HTTP module, use the require() method:

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

Node.js as a Web Server:

- The HTTP module can create an HTTP server that listens to server ports and gives a response back to the client.

Use the createServer() method to create an HTTP server:

Example: //this is how a basic server is implemented

```
var http = require('http');
//create a server object:
http.createServer(function (req, res) {
  res.write('Hello World!'); //write a response to the client
  res.end(); //end the response
}).listen(8080); //the server object listens on port 8080
```

- The function passed into the http.createServer() method, will be executed when someone tries to access the computer on port 8080.
- Save the code above in a file called "demo_http.js", and initiate the file:
- Initiate demo_http.js: C:\Users\Your Name>node demo_http.js
- If you have followed the same steps on your computer, you will see the same result as the example: http://localhost:8080

When you are building a web application you have generally two kinds of things that the server sends to the client (browser).

1. Static Files: These are files that don't change .They aren't dependant on any user input generally.
2. Dynamic Files :
    - A webserver may need to query a database and return variable things like records of information etc that is dependent on some type of user input.
    - These are considered dynamic because they can change based on whatever the application is doing.

<u>Serving Static Files:</u>

- Beyond normal browser usage, there are thousands of other reasons you'd need to serve a static files, like for downloading music or scientific data. Either way, you'll need to come up with a simple way to let the user download these files from your server.
- One of the most fundamental uses of an HTTP server is to serve static files to a user's browser, like CSS, JavaScript, or image files.
- One simple way to do this is to create a Node HTTP server. As you probably know, Node.js excels at handling I/O-intensive tasks, which makes it a natural choice here.
- You can choose to create your own simple HTTP server from the base http module that's shipped with Node, or you can use the popular serve-static package, which provides many common features of a static file server.
- The end goal of our static server is to let the user specify a file path in the URL and have that file returned as the contents of the page. However, the user shouldn't be able to specify just any path on our server, otherwise a malicious user could try to take advantage of a misconfigured system and steal sensitve information.
- A simple attack might look like this: localhost:8080/etc/shadow. Here the attacker would be requesting the /etc/shadow file. To prevent these kinds of attacks, we should be able to tell the server to only allow the user to download certain files, or only files from certain directories (like /var/www/my-website/public).

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.

Listing 7.1 `http_server_static.js`: **Implementing a basic static file webserver**

```
01 var fs = require('fs');
02 var http = require('http');
03 var url = require('url');
04 var ROOT_DIR = "html/";
05 http.createServer(function (req, res) {
06   var urlObj = url.parse(req.url, true, false);
07   fs.readFile(ROOT_DIR + urlObj.pathname, function (err,data) {
08     if (err) {
09       res.writeHead(404);
10       res.end(JSON.stringify(err));
11       return;
12     }
13     res.writeHead(200);
14     res.end(data);
15   });
16 }).listen(8080);
```

- In the above code, line 5 creates the server using createServer() and also defines the request event handler shown in lines 6–15.
- The server is listening on port 8080 by calling listen() on the Server object. Inside the request event handler on line 6, the url.parse() method is used to parse the url so that we can use the pathname attribute when specifying the path for the file in line 7.
- The static file is opened and read using fs.readFile(), and in the readFile() callback the contents of the file are written to the response object using res.end(data) on line 14.

Listing 7.2 `http_client_static.js`: **Basic web client retrieving static files**

```
01 var http = require('http');
02 var options = {
03     hostname: 'localhost',
04     port: '8080',
05     path: '/hello.html'
06   };
07 function handleResponse(response) {
08   var serverData = '';
09   response.on('data', function (chunk) {

10     serverData += chunk;
11   });
12   response.on('end', function () {
13     console.log(serverData);
14   });
15 }
16 http.request(options, function(response){
17   handleResponse(response);
18 }).end();
```

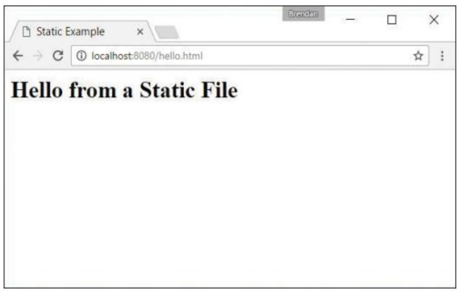
Listing 7.2 shows a basic implementation of an HTTP client that sends a get request to the server to retrieve the file contents.

- The options for the request are set in lines 2–6, and then the client request is initiated in lines 16–18 passing the options.

- When the request completes, the callback function uses the on('data') handler to read the contents of the response from the server and then the on('end') handler to log the file contents to a file.

Listing 7.2 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>
```

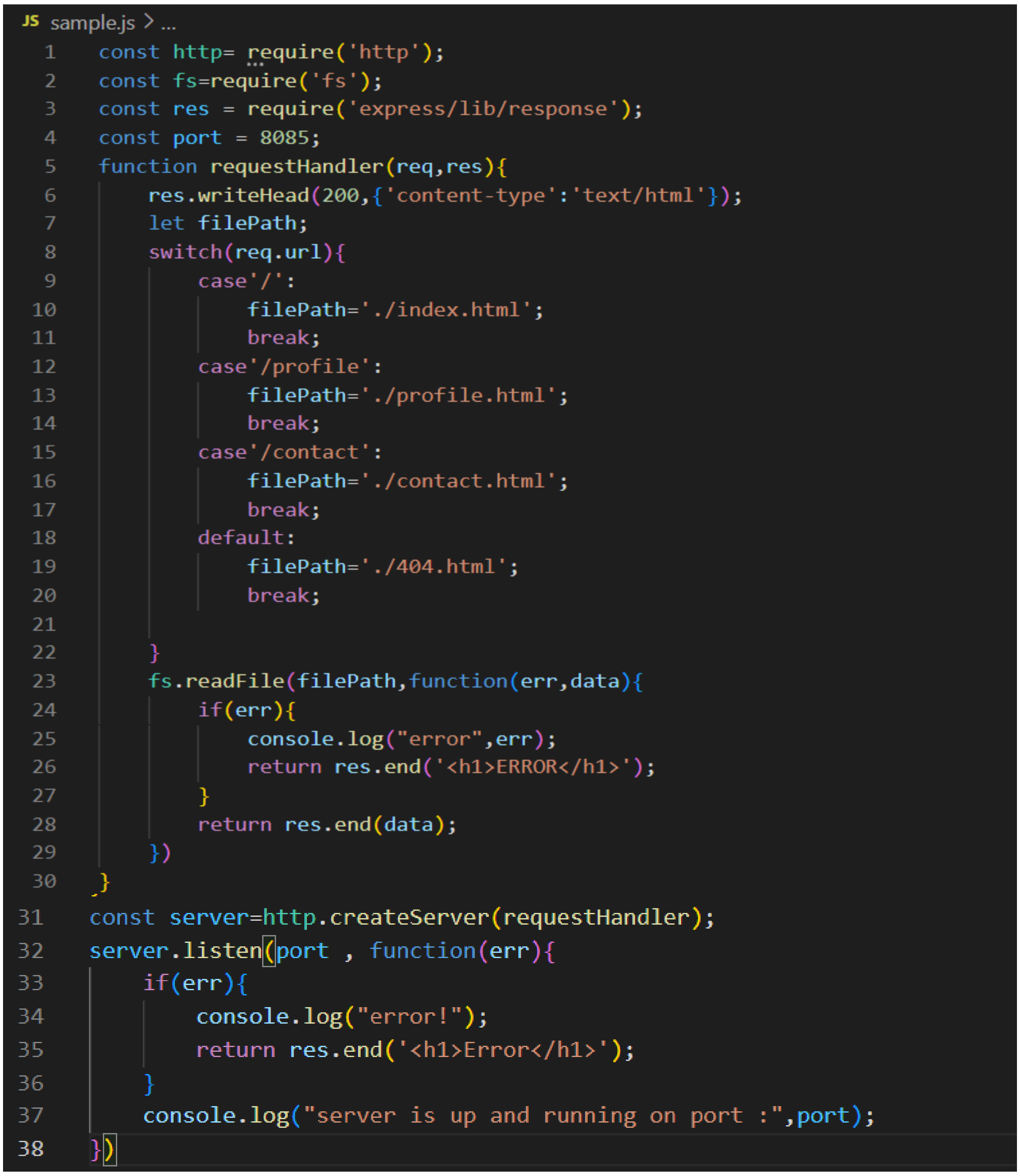


Hello from a Static File

Figure 7.2 Implementing a basic static file web server

Another example for accessing static files is as follows:

We can view various pages related to a website using http server.



sample.js > ...

```javascript
const http= require('http');
const fs=require('fs');
const res = require('express/lib/response');
const port = 8085;
function requestHandler(req,res){
    res.writeHead(200,{'content-type':'text/html'});
    let filePath;
    switch(req.url){
        case'/':
            filePath='./index.html';
            break;
        case'/profile':
            filePath='./profile.html';
            break;
        case'/contact':
            filePath='./contact.html';
            break;
        default:
            filePath='./404.html';
            break;

    }
    fs.readFile(filePath,function(err,data){
        if(err){
            console.log("error",err);
            return res.end('<h1>ERROR</h1>');
        }
        return res.end(data);
    })
}
const server=http.createServer(requestHandler);
server.listen(port , function(err){
    if(err){
        console.log("error!");
        return res.end('<h1>Error</h1>');
    }
    console.log("server is up and running on port :",port);
})
```

- This Js file code enables us to access various html pages using http server.

Exmaple-2: //app.js

```javascript
const http = require('http');
// File system module used for accessing files in nodejs
const fs = require("fs");
const port = 3000;
  // Helper function
```

```
function readAndServe(path, res)
{
    fs.readFile(path,function(err, data)
    {
        console.log(data);

        // res.setHeader('Content-Type', 'text/html');
        res.end(data);
    })
}
const server = http.createServer((req, res) => {
const url = req.url;
const method = req.method;
/* Serving static files on specific Routes */
if(url === "/")
{
    readAndServe("./index.html",res)
  /* The file named index.html will be sent as a response when the index url
is requested */
}
else if(url === "/about")
{
    readAndServe("./about.html",res)
    /*The about.html file will be sent as a response when /about is requested
*/
}
else
{
    res.end("Path not found");
    /* All paths other than / and /about will send an error as a response */
  }
});
server.listen(port, () => {
  console.log(`Server running at http://:${port}/`);
});
```

//index.html:

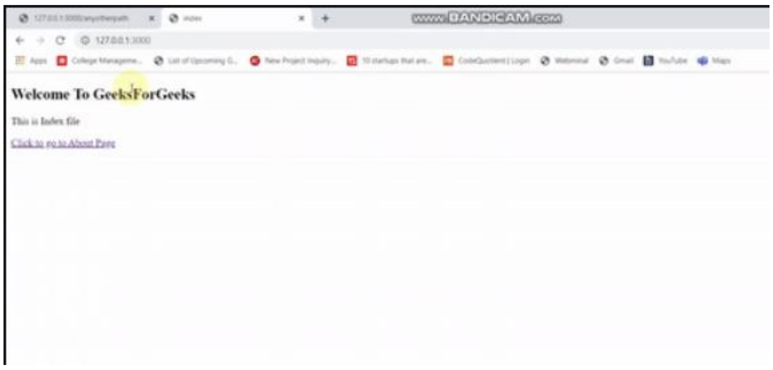
```
<!DOCTYPE html>
<!DOCTYPE html>
<html>
  <head>
    <title>index</title>
  </head>
  <body>
    <h2>Welcome To GeeksForGeeks</h2>
    <p>This is Index file</p>
    <p><a href="/about">
```

```
        Click to go to About Page
      </a>
    </p>
  </body>
</html>
```

//about.html:

```
<!DOCTYPE html>
<!DOCTYPE html>
<html>
  <head>
    <title>index</title>
  </head>
  <body>
    <h2>Welcome To GeeksForGeeks</h2>
    <p>This is Index file</p>
    <p><a href="/about">
        Click to go to About Page
      </a>
    </p>
  </body>
</html>
```

Output:



Welcome To GeeksForGeeks

This is Index file

Click to go to About Page

Implementing Dynamic GET Servers:

- More often than not you will use Node.js webservers to serve dynamic content rather than static content.
- This content may be dynamic HTML files or snippets, JSON data, or a number of other data types.
- To serve a GET request dynamically, you need to implement code in the request handler that dynamically populates the data you want to send back to the client, writes it out to the

response, and then calls end() to finalize the response and flush the Writable stream.

Listing 7.3 `http_server_get.js`: **Implementing a basic `GET` webserver**

```
01 var http = require('http');
02 var messages = [
03   'Hello World',
04   'From a basic Node.js server',
05   'Take Luck'];
06 http.createServer(function (req, res) {
07   res.setHeader("Content-Type", "text/html");
08   res.writeHead(200);
09   res.write('<html><head><title>Simple HTTP Server</title></head>');
10   res.write('<body>');
11   for (var idx in  messages){
12     res.write('\n<h1>' + messages[idx] + '</h1>');
13   }
14   res.end('\n</body></html>');
15 }).listen(8080);
```

Listing 7.3 shows the basic implementation of a dynamic web service. In this case, the web service simply responds with a dynamically generated HTTP file. The example is designed to show the process of sending the headers, building the response, and then sending the data in a series of write() requests.

- The line 6 creates the server using createServer(), and line 15 begins listening on port 8080 using listen().
- Inside the request event handler defined in lines 7–15, the Content-Type header is set and then the headers are sent with a response code of 200. In reality you would have already done a lot of processing to prepare the data. But in this case, the data is just the messages array defined in lines 2–5.
- In lines 11–13 the loop iterates through the messages and calls write() each time to stream the response to the client. Then in line 14 the response is completed by calling end().

Listing 7.4 `http_client_get.js`: **Basic web client that makes a `GET` request to the server in Listing 7.3**

```
01 var options = {
02     hostname: 'localhost',
03     port: '8080',
04   };
05 function handleResponse(response) {
06   var serverData = '';
07   response.on('data', function (chunk) {
08     serverData += chunk;
09   });
10   response.on('end', function() {
11     console.log("Response Status:", response.statusCode);
12     console.log("Response Headers:", response.headers);
13     console.log(serverData);
14   });
15 }
16 http.request(options, function(response){
17   handleResponse(response);
18 }).end
```

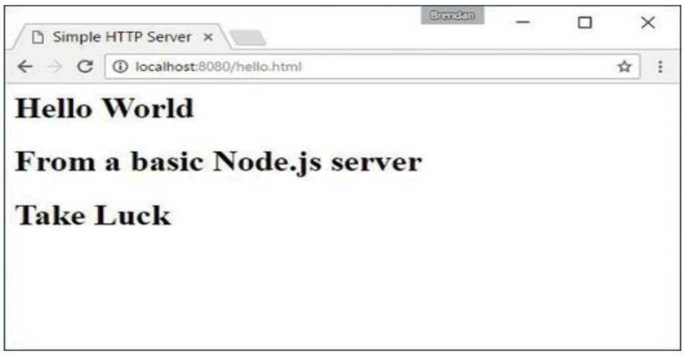
Listing 7.4 shows a basic implementation of an HTTP client that reads the response from the server in Listing 7.3. This is similar to the example in Listing 7.2; however, note that no path was specified since the service doesn't really require one. For more complex services, you would implement query strings or complex path routes to handle a variety of calls.

- On line 11 the statusCode from the response is logged to the console.
- Also on line12 the headers from the response are also logged. Then on line 13 the full response from the server is logged.

Figure 7.3 and Listing 7.4 Output show the output of the HTTP client as well as accessing the dynamic get server from a web browser.

Listing 7.4 Output **Implementing a basic HTTP GET service**

```
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>
```



Hello World

From a basic Node.js server

Take Luck

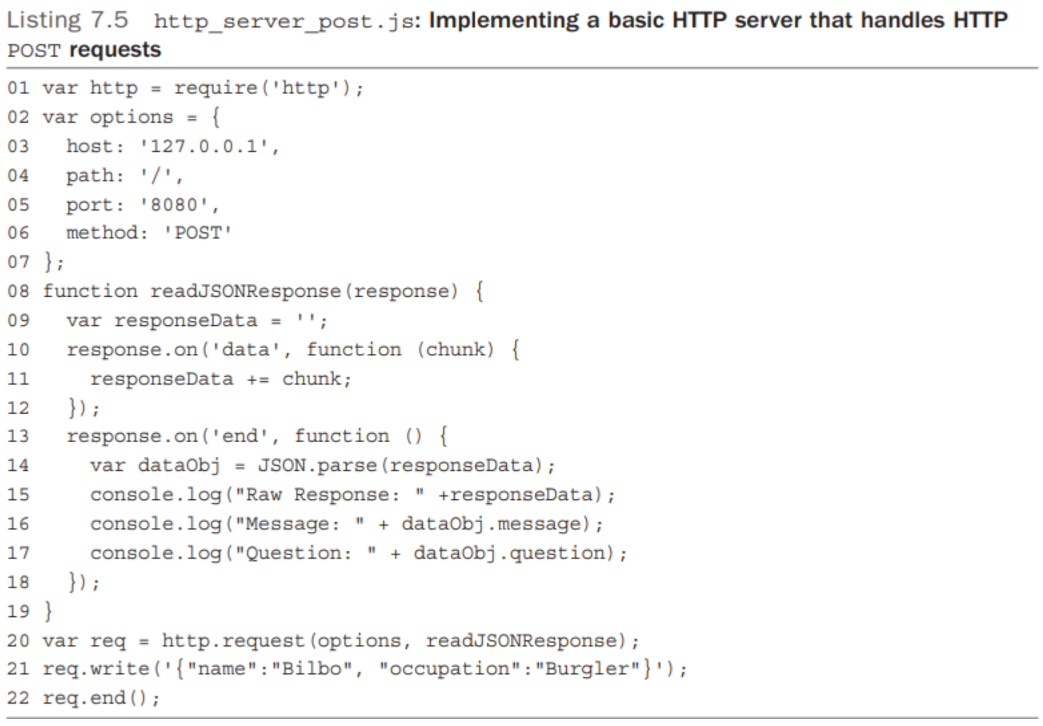
Figure 7.3 Output of a basic HTTP GET server

Implementing POST Servers :

Implementing a POST service is similar to implementing a GET server. In fact, you may end up implementing them together in the same code for the sake of convenience.

- POST services are handy if you need to send data to the server to be updated, as for form submissions.
- To serve a POST request, you need to implement code in the request handler that reads the contents of the post body out and processes it.
- Once you have processed the data, you dynamically populate the data you want to send back to the client, write it out to the response, and then call end() to finalize the response and flush the Writable stream.
- Just as with a dynamic GET server, the output of a POST request may be a webpage, HTTP snippet, JSON data, or some other data.

Listing 7.5 shows the basic implementation of a dynamic web service handling POST requests. In this case, the web service accepts a JSON string from the client representing an object that has name and occupation properties. The code in lines 4–6 read the data from the request stream, and then in the event handler in lines 7–14, the data is converted to an object and used to build a new object with message and question properties. Then in line 14 the new object is stringified and sent back to the client in the end() call.



Listing 7.5 `http_server_post.js`: **Implementing a basic HTTP server that handles HTTP** `POST` **requests**

```
01 var http = require('http');
02 var options = {
03   host: '127.0.0.1',
04   path: '/',
05   port: '8080',
06   method: 'POST'
07 };
08 function readJSONResponse(response) {
09   var responseData = '';
10   response.on('data', function (chunk) {
11     responseData += chunk;
12   });
13   response.on('end', function () {
14     var dataObj = JSON.parse(responseData);
15     console.log("Raw Response: " +responseData);
16     console.log("Message: " + dataObj.message);
17     console.log("Question: " + dataObj.question);
18   });
19 }
20 var req = http.request(options, readJSONResponse);
21 req.write('{"name":"Bilbo", "occupation":"Burgler"}');
22 req.end();
```

- Listing 7.6 shows a basic implementation of an HTTP client that sends JSON data to the server as part of a POST request.
- The request is started in line 20. Then in line 21 a JSON string is written to the request stream, and line 22 finishes the request with end().
- Once the server sends the response back, the on('data') handler in lines 10–12 reads the JSON response.
- Then the on('end') handler in lines 13–18 parses the response into a JSON object and outputs the raw response, message, and question.
- Output 7.6 shows the output of the HTTP POST client.

Listing 7.6 `http_client_post.js`: **Basic HTTP client that sends JSON data to the server using POST and handles the JSON response**

```
var http = require('http');
var options = {
  host: '127.0.0.1',
  path: '/',
  port: '8080',
  method: 'POST'
};
function readJSONResponse (response) {
  var responseData = '';
  response.on('data', function (chunk) {
    responseData += chunk;
  });
  response.on('end', function () {
    var dataObj = JSON.parse(responseData);
    console.log("Raw Response: " +responseData);
    console.log("Message: " + dataObj.message);
    console.log("Question: " + dataObj.question);
  });
}
var req = http.request(options, readJSONResponse);
req.write('{"name":"Bilbo", "occupation":"Burgler"}');
req.end();
```

Listing 7.6 Output **Implementing an HTTP POST server serving JSON data**

```
C:\books\node\ch07>node http_server_post.js
Raw Response: {"message":"Hello Bilbo","question":"Are you a good Burgler?"}
Message: Hello Bilbo
Question: Are you a good Burgler?
```

<u>Interacting with External Sources:</u>

A common use of the HTTP services in Node.js is to access external systems to get data to fulfill client requests. A variety of external systems provide data that can be used in various ways.

In this example, the code connects to the openweathermap.org API to retrieve weather information about a city.

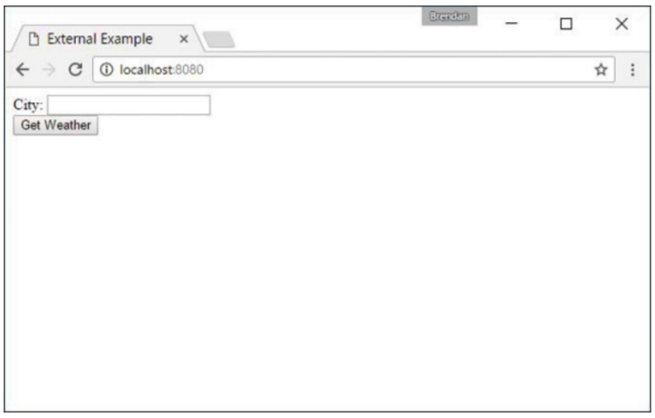
- To keep the example simple, the output from openweathermap.org is pushed to the browser in a raw format. In reality, you would likely massage the pieces of data needed into your own pages, widgets, or data responses.
- Listing 7.7 shows the implementation of the web service that accepts both GET and POST requests. For the GET request, a simple webpage with a form is returned that allows the user to post a city name.
- Then in the POST request the city name is accessed, and the Node.js web client starts up and connects remotely to openweathermap.org to retrieve weather information for that city.
- Then that info is returned to the server along with the original web form. The big difference between this example and the previous examples is that the webserver also implements a local web client to connect to the external service and get data used to formulate the response.
- The webserver is implemented in lines 35–49. If the method is POST, we read the form data from the request stream and use querystring.parse() to get the city name and call into the getWeather() function.

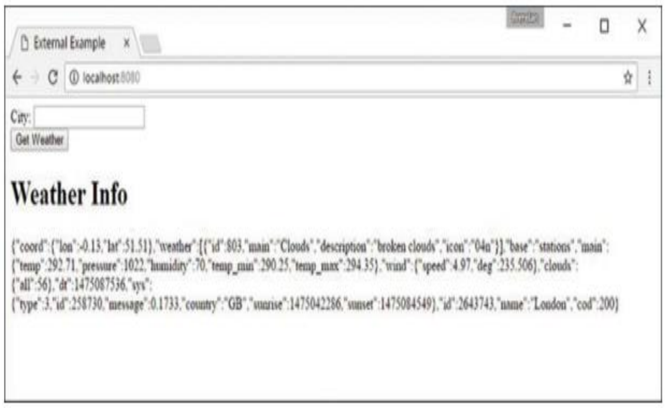
- The getWeather() function in lines 27–33 implements the client request to openweathermap.org. Then the parseWeather() request handler in lines 17–25 reads the response from openweathermap.org and passes that data to the sendResponse() function defined in lines 4–16 that formulates the response and sends it back to the client.

Figure 7.4 shows the implementation of the external service in a web browser.

Listing 7.7 `http_server_external`: **Implementing an HTTP web service that connects remotely to an external source for weather data**

```
var http = require('http');
var url = require('url');
var qstring = require('querystring');
var APIKEY = ""//place your own api key within the quotes;
function sendResponse(weatherData, res){
  var page = '<html><head><title>External Example</title></head>' +
    '<body>' +
    '<form method="post">' +
    'City: <input name="city"><br>' +
    '<input type="submit" value="Get Weather">' +
    '</form>';
  if(weatherData){
    page += '<h1>Weather Info</h1><p>' + weatherData +'</p>';
  }
  page += '</body></html>';
  res.end(page);
}
function parseWeather(weatherResponse, res) {
  var weatherData = '';
  weatherResponse.on('data', function (chunk) {
    weatherData += chunk;
  });
  weatherResponse.on('end', function () {
    sendResponse(weatherData, res);
  });
}
function getWeather(city, res){
  city = city.replace(' ', '-');
  console.log(city);
  var options = {
    host: 'api.openweathermap.org',
    path: '/data/2.5/weather?q=' + city + '&APPID=' + APIKEY
  };
  http.request(options, function(weatherResponse){
    parseWeather(weatherResponse, res);
  }).end();
}
http.createServer(function (req, res) {
  console.log(req.method);
  if (req.method == "POST"){
    var reqData = '';
    req.on('data', function (chunk) {
      reqData += chunk;
    });
    req.on('end', function() {
      var postParams = qstring.parse(reqData);
      getWeather(postParams.city, res);
    });
  } else {
    sendResponse(null, res);
  }
}).listen(8080);
```





Weather Info

```
{"coord":{"lon":-0.13,"lat":51.51},"weather":[{"id":803,"main":"Clouds","description":"broken clouds","icon":"04n"}],"base":"stations","main":
{"temp":292.71,"pressure":1022,"humidity":70,"temp_min":290.25,"temp_max":294.35},"wind":{"speed":4.97,"deg":235.506},"clouds":
{"all":56},"dt":1475087536,"sys":
{"type":3,"id":258730,"message":0.1733,"country":"GB","sunrise":1475042286,"sunset":1475084549},"id":2643743,"name":"London","cod":200}
```

Implementing HTTPS Servers and Clients:

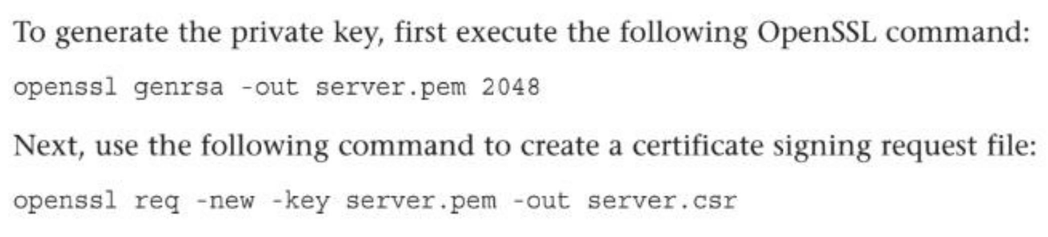
The **HTTP protocol** is one of the most important protocols for smooth communication between the networks but in the HTTP protocol, data is not encrypted so any sensitive information can be sniffed during the communication so there is another version of the HTTP i .e **HTTPS** that encrypts the data during the transmission between a browser and the server.

Hypertext Transfer Protocol Secure (HTTPS) is a communications protocol that provides secure communication between HTTP clients and servers. HTTPS is really just HTTP running on top of the SSL/TLS protocol, which is where it gets its security capabilities.

HTTP provides security in two main ways.

- First, It uses long-term public and secret keys to exchange a short-term session key so that data can be encrypted between client and server.
- Second, It provides authentication so that you can ensure that the webserver you are connecting to is the one you actually think it is, thus preventing man-in-the-middle attacks where requests are rerouted through a third party.

Before getting started using HTTPS, you need to generate a private key and a public certificate. There are several ways to do this, depending on your platform. One of the simplest methods is to use the OpenSSL library for your platform.

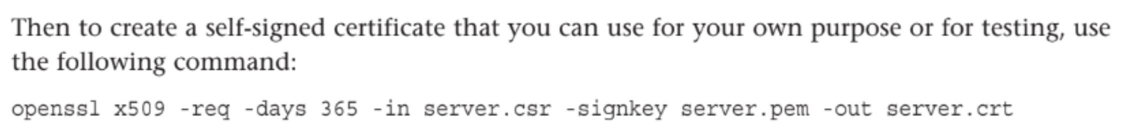


To generate the private key, first execute the following OpenSSL command:

```
openssl genrsa -out server.pem 2048
```

Next, use the following command to create a certificate signing request file:

```
openssl req -new -key server.pem -out server.csr
```



Then to create a self-signed certificate that you can use for your own purpose or for testing, use the following command:

```
openssl x509 -req -days 365 -in server.csr -signkey server.pem -out server.crt
```

<u>Creating an HTTPS Client:</u>

Creating an HTTPS client is almost exactly like the process of creating an HTTP client

The most important options you really need to worry about are **key, cert,** and **agent.**

- The key option specifies the private key used for SSL.
- The cert value specifies the x509 public key to use.
- The global agent does not support options needed by HTTPS, so you need to disable the agent by setting the agent to null, as shown here:

```
var options = {
  key: fs.readFileSync('test/keys/client.pem'),
  cert: fs.readFileSync('test/keys/client.crt),
  agent: false
};
```

You can also create your own custom Agent object that specifies the agent options used for the request:

```
options.agent = new https.Agent (options);
```

Once you have defined the options with the cert, key, and agent settings, you can call the

https.request(options, [responseCallback]), and it will work exactly the same as the

http.request() call. The only difference is that the data between the client and server is encrypted.

```
var options = {
  hostname: 'encrypted.mysite.com',
  port: 443,
  path: '/',
  method: 'GET',
  key: fs.readFileSync('test/keys/client.pem'),
  cert: fs.readFileSync('test/keys/client.crt),
  agent: false
};
var req = https.request(options, function(res)) {
  <handle the response the same as an http.request>
}
```

Creating an HTTPS Server:

Creating an HTTPS server is almost exactly like the process of creating an HTTP server The only difference is that there are additional options parameters that you must pass into https.createServer() The most important options you really need to worry about are

key and **cert.**

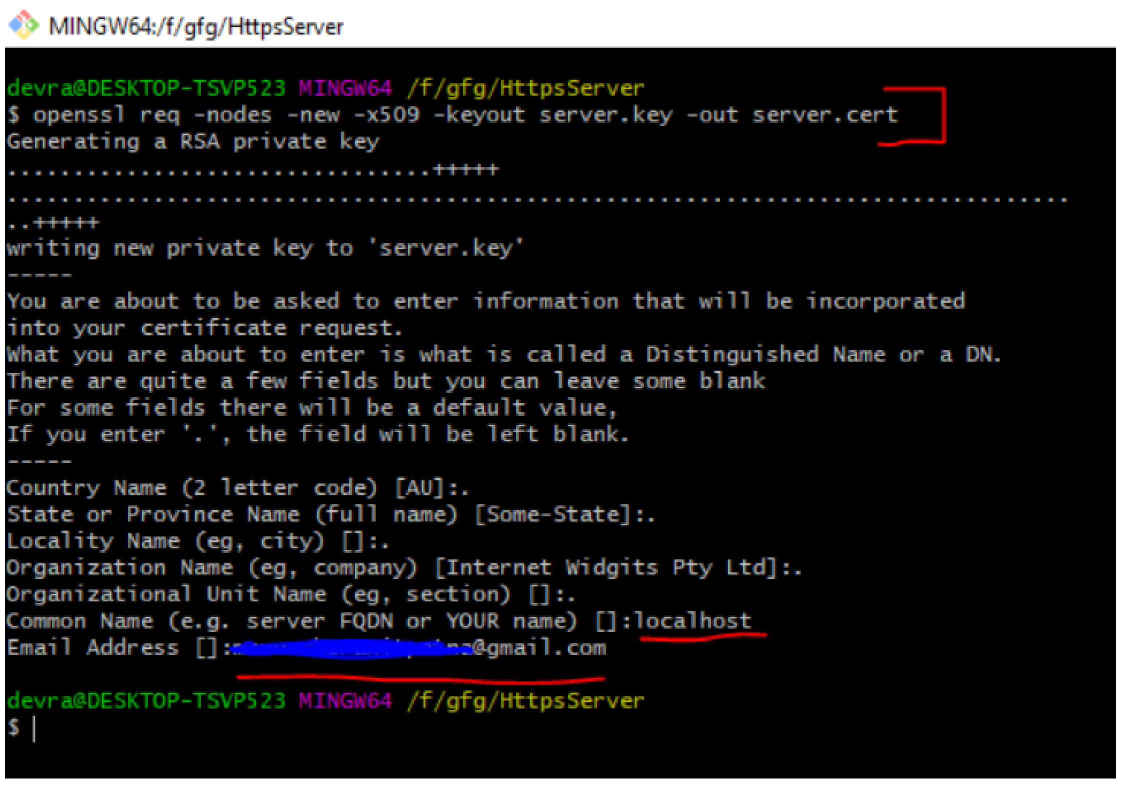
- The key option specifies the private key used for SSL.
- The cert value specifies the x509 public key to use.

How to create HTTPS Server with Node.js ?

To built an HTTPS server with node js, we need an SSL (Secure Sockets Layer) certificate. We can create a self-signed SSL certificate on our local machine. Let's first create an SSL certificate on our machine first.

Step 1: First of all we would generate a self-signed certificate. Open your terminal or git bash and run the following command:

```
openssl req -nodes -new -x509 -keyout server.key -out server.cert
```



MINGW64:/f/gfg/HttpsServer

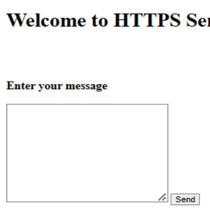
```
devra@DESKTOP-TSVP523 MINGW64 /f/gfg/HttpsServer
$ openssl req -nodes -new -x509 -keyout server.key -out server.cert
Generating a RSA private key
................................+++++
................................................................................
..+++++
writing new private key to 'server.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:.
State or Province Name (full name) [Some-State]:.
Locality Name (eg, city) []:.
Organization Name (eg, company) [Internet Widgits Pty Ltd]:.
Organizational Unit Name (eg, section) []:.
Common Name (e.g. server FQDN or YOUR name) []:localhost
Email Address []:[redacted]@gmail.com

devra@DESKTOP-TSVP523 MINGW64 /f/gfg/HttpsServer
$ |
```

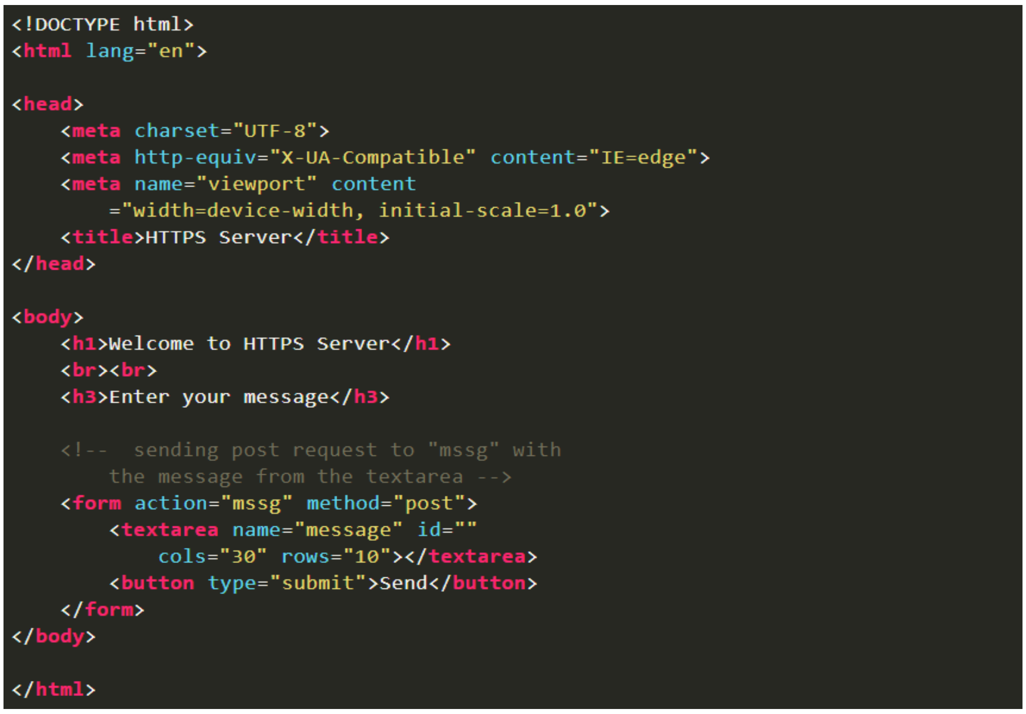
creating SSL Certificate

This would generate two files:

- server.cert: The self-signed certificate file.
- server.key: The private key of the certificate.

<u>Step 2</u>: Now let’s code the index.html file. We will create a form to send a message to the server through a POST request.



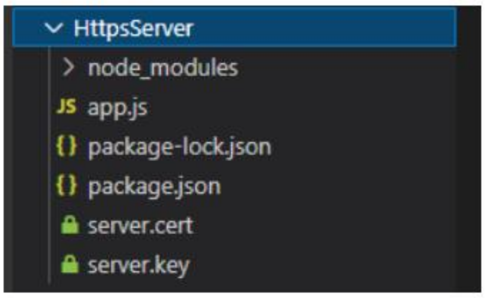


```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content
        ="width=device-width, initial-scale=1.0">
    <title>HTTPS Server</title>
</head>

<body>
    <h1>Welcome to HTTPS Server</h1>
    <br><br>
    <h3>Enter your message</h3>

    <!--  sending post request to "mssg" with
        the message from the textarea -->
    <form action="mssg" method="post">
        <textarea name="message" id=""
            cols="30" rows="10"></textarea>
        <button type="submit">Send</button>
    </form>
</body>

</html>
```

Step 3: Now create an app.js file. We would initialize the project using npm in the terminal

```
npm init
```

Project Structure:



```
HttpsServer
  node_modules
  app.js
  package-lock.json
  package.json
  server.cert
  server.key
```

Step 4: Now we will code the app.js file. In this file, we create an HTTPS server using createServer() function. We pass the certificate and key files of the SSL certificate as options object in createServer() function. We handle GET and POST requests using express in NodeJs.

```javascript
// Requiring in-built https for creating
// https server
const https = require("https");

// Express for handling GET and POST request
const express = require("express");
const app = express();

// Requiring file system to use local files
const fs = require("fs");

// Parsing the form of body to take
// input from forms
const bodyParser = require("body-parser");

// Configuring express to use body-parser
// as middle-ware
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

// Get request for root of the app
app.get("/", function (req, res) {

  // Sending index.html to the browser
  res.sendFile(__dirname + "/index.html");
});

// Post request for geetting input from
// the form
app.post("/mssg", function (req, res) {

  // Logging the form body
  console.log(req.body);

  // Redirecting to the root
  res.redirect("/");
});

// Creating object of key and certificate
// for SSL
const options = {
  key: fs.readFileSync("server.key"),
  cert: fs.readFileSync("server.cert"),
};

// Creating https server by passing
// options and app object
https.createServer(options, app)
```

```
.listen(3000, function (req, res) {
  console.log("Server started at port 3000");
});
```

Step 5: Run node app.js file

Now open the browser and type the running server address:

https://localhost:3000/

Now you would see a webpage running with HTTPS. Write your message in the text area.

Now hit the send button and see it in your console. The output would be:

So, In this way, we can create an HTTPS server using Node.js

Implementing Express in Node.Js:

- Getting started with Express
- Configuring Routes
- Using Request/Response Objects

- Express is a lightweight module that wraps the functionality of the Node.js http module in a simple to use interface.
- Express also extends the functionality of the http module to make it easy for you to handle server routes, responses, cookies, and statuses of HTTP requests.

Getting Started with Express:

- It is simple to start using Express in your Node.js projects.
- All you need to do is add the express module using the following command from the root of your project:

```
npm install express
```

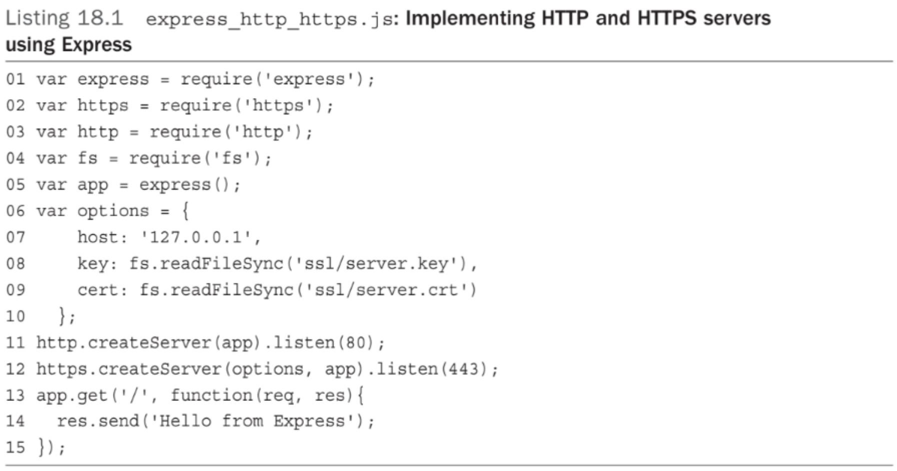
- You can also add express to your package.json module to ensure that express is installed when you deploy your application.
- Once you have installed express, you need to create an instance of the express class to act as the HTTP server for your Node.js application.
- The following lines of code import the express module and create an instance of express that you can use:

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

var app = express();
```

- To begin implementing Express as the HTTP server for your Node.js application, you need to create an instance and begin listening on a port.
- The following three lines of code start a rudimentary Express server listening on port 8080:

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

var app = express();

app.listen(8080);
```

- The app.listen(port) call binds the underlying HTTP connection to the port and begins listening on it.
- The underlying HTTP connection is the same connection produced using the listen() method on a Server object created using the http library discussed earlier in this book.
- In fact, the value returned by express() is actually a callback function that maps to the callback function that is passed into the http.createServer() and https.createServer() methods.



Listing 18.1 express_http_https.js: **Implementing HTTP and HTTPS servers using Express**

```
var express = require('express');
var https = require('https');
var http = require('http');
var fs = require('fs');
var app = express();
var options = {
    host: '127.0.0.1',
    key: fs.readFileSync('ssl/server.key'),
    cert: fs.readFileSync('ssl/server.crt')
  };
http.createServer(app).listen(80);
https.createServer(options, app).listen(443);
app.get('/', function(req, res){
  res.send('Hello from Express');
});
```

<u>Application</u>:

- ➢ The app object conventionally denotes the Express application.
- ➢ Create it by calling the top-level express() function exported by the Express module:

```
const express = require('express');
const app = express( );
app.get('/', (req, res) => {
   res.send('hello world');
})
App.listen(3000);
```

- ➢ The app object has methods for:
  - Routing HTTP requests; for example app.METHOD
  - Configuring middleware; for example app.route
  - Rendering HTML views; for example app.render
  - Registering a template engine; for example app.engine

- ➢ There are two parts when defining the route. First is the HTTP request method (typically GET or POST). Second, is the path specified in the URL—for example, / for the root of the website, /login for the login page, and /cart to display a shopping cart.
- ➢ The express module provides a series of functions that allow you to implement routes for the Express server. These functions all use the following syntax:

app.<method>(path, [callback . . .], callback)

- ➢ The <method> portion of the syntax actually refers to the HTTP request method, such as GET or POST. For example:

app.get(path, [middleware, ...], callback)

app.post(path, [middleware, ...], callback)

- ➢ The path refers to the path portion of the URL that you want to be handled by the callback function.
- ➢ The middleware parameters are 0 or more middleware functions that are applied before executing the callback function.
- ➢ The callback function is the request handler that handles the request and sends the response back to the client.

- The callback function accepts a Request object as the first parameter and a Response object as the second.

For example, the following code implements some basic GET and POST routes:

```
app.get('/', function(req, res){
   res.send("Server Root");
});
app.get('/login', function(req, res){
   res.send("Login Page");
});
app.post('/save', function(req, res){
   res.send("Save Page");
});
```

- Express also provides the app.all() method that works exactly the same as the app.post() and app.get() methods.
- The only difference is that the callback function for app.all() is called on every request for the specified path regardless of HTTP method.
- Also, the app.all() method can accept the * character as a wildcard in the path.
- This is a great feature for implementing request logging or other special functionality to handle requests.

Configuring Routes:

➢ To reduce the number of routes, you can implement parameters within the URL. Parameters allow you to use the same route for similar requests by providing unique values for different requests that define how your application handles the request and builds the response.

There are four main methods for implementing parameters in a route:

■ **Query string**: Uses the standard ?key=value&key=value... HTTP query string after the path in the URL. This is the most common method for implementing parameters, but the URLs can become long and convoluted.

■ **POST params**: When implementing a web form or other POST request, you can pass parameters in the body of the request.

■ **regex**: Defines a regular expression as the path portion of the route. Express uses the regex to parse the path of the URL and store matching expressions as an array of parameters.

■ **Defined parameter**: Defines a parameter by name using :<param_name> in the path portion of the route. Express automatically assigns that parameter a name when parsing the path.

Applying Route Parameters Using Query Strings

```
app.get('/find', function(req, res){
   var url_parts = url.parse(req.url, true);
   var query = url_parts.query;
   res.send('Finding Book: Author: ' + query.author +' Title: ' + query.title);
});
```

For example, consider the following URL:

```
/find?author=Brad&title=Node
------------------------------------------------------------
```

The res.send() method returns:

```
Finding Book: Author: Brad Title: Node
```

Applying Route Parameters Using Regex

```
app.get(/^\/book\/(\w+)\:(\w+)?$/, function(req, res){
   res.send('Get Book: Chapter: ' + req.params[0] +' Page: ' + req.params[1]);
});
```

For example, consider the following URL:

```
/book/12:15
------------------------------------------------------------
```

The res.send() method returns

```
Get Book: Chapter: 12 Page: 15
```

Applying Route Parameters Using Defined Parameters

```
app.get('/user/:userid', function (req, res) {
   res.send("Get User: " + req.param("userid"));
});
```

For example, consider the following URL:

/user/4983

The res.send() method returns

```
Get User: 4983
```

Applying Callback Functions for Defined Parameters

```
app.param('userid', function(req, res, next, value){
   console.log("Request with userid: " + value);
   next();
});
```

To see how the preceding code works, consider the following URL:

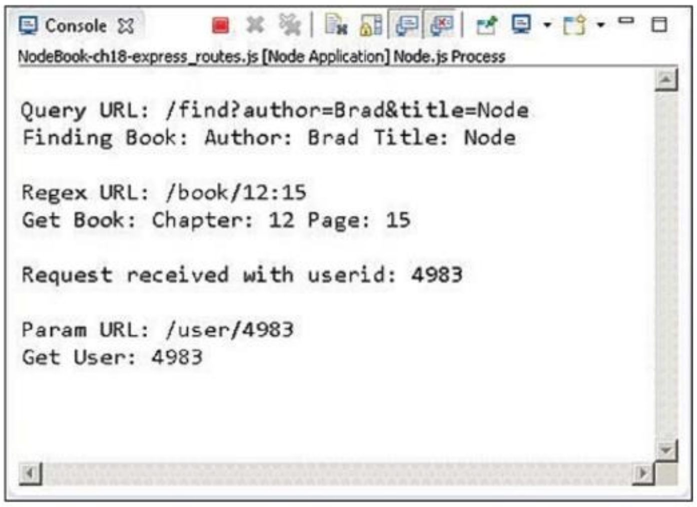
/user/4983

The userid of 4983 is parsed from the URL and the consol.log() statement displays

```
Request with userid: 4983
```

Listing 18.2 `express_routes.js`: **Implementing route parameters in Express**

```
var express = require('express');
var url = require('url');
var app = express();
app.listen(80);
app.get('/', function (req, res) {
  res.send("Get Index");
});
app.get('/find', function(req, res){
  var url_parts = url.parse(req.url, true);
  var query = url_parts.query;
  var response = 'Finding Book: Author: ' + query.author +
                      ' Title: ' + query.title;
  console.log('\nQuery URL: ' + req.originalUrl);
  console.log(response);
  res.send(response);
});
app.get(/^\/book\/(\w+)\:(\w+)?$/, function(req, res){
  var response = 'Get Book: Chapter: ' + req.params[0] +
                   ' Page: ' + req.params[1];
  console.log('\nRegex URL: ' + req.originalUrl);
  console.log(response);
  res.send(response);
});
app.get('/user/:userid', function (req, res) {
  var response = 'Get User: ' + req.param('userid');
  console.log('\nParam URL: ' + req.originalUrl);
  console.log(response);
  res.send(response);
});
app.param('userid', function(req, res, next, value){
  console.log("\nRequest received with userid: " + value);
  next();
});
```


Console
NodeBook-ch18-express_routes.js [Node Application] Node.js Process

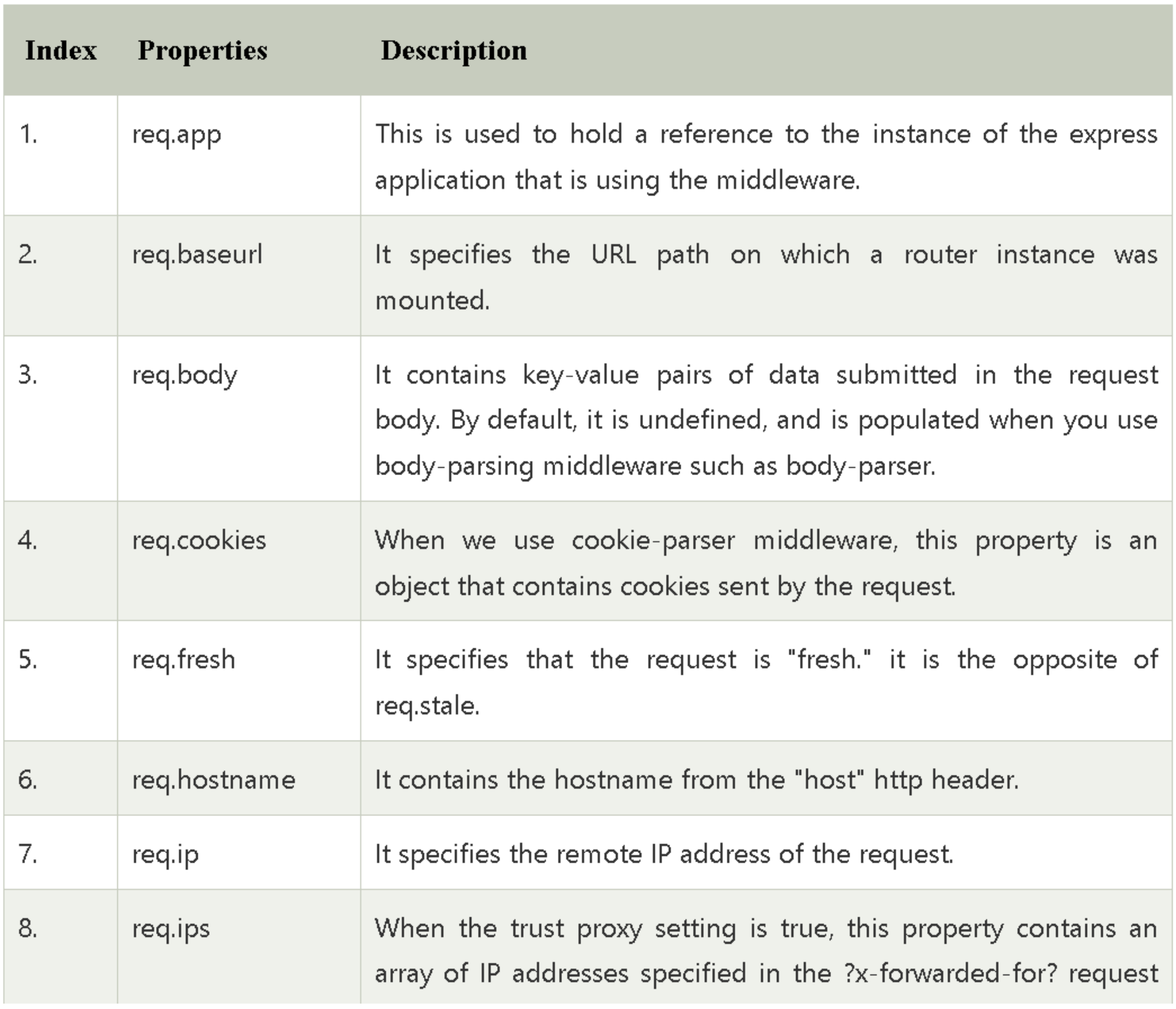
```

Query URL: /find?author=Brad&title=Node
Finding Book: Author: Brad Title: Node

Regex URL: /book/12:15
Get Book: Chapter: 12 Page: 15

Request received with userid: 4983

Param URL: /user/4983
Get User: 4983
```

Using Request and Response Objects:

REQUEST OBJECTS:

- Express.js request and response objects are the parameters of the callback function which is used in Express applications.
- The express.js request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.

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



Index	Properties	Description
1.	req.app	This is used to hold a reference to the instance of the express application that is using the middleware.
2.	req.baseurl	It specifies the URL path on which a router instance was mounted.
3.	req.body	It contains key-value pairs of data submitted in the request body. By default, it is undefined, and is populated when you use body-parsing middleware such as body-parser.
4.	req.cookies	When we use cookie-parser middleware, this property is an object that contains cookies sent by the request.
5.	req.fresh	It specifies that the request is "fresh." it is the opposite of req.stale.
6.	req.hostname	It contains the hostname from the "host" http header.
7.	req.ip	It specifies the remote IP address of the request.
8.	req.ips	When the trust proxy setting is true, this property contains an array of IP addresses specified in the ?x-forwarded-for? request

		header.
9.	req.originalurl	This property is much like req.url; however, it retains the original request URL, allowing you to rewrite req.url freely for internal routing purposes.
10.	req.params	An object containing properties mapped to the named route ? parameters?. For example, if you have the route /user/:name, then the "name" property is available as req.params.name. This object defaults to {}.
11.	req.path	It contains the path part of the request URL.
12.	req.protocol	The request protocol string, "http" or "https" when requested with TLS.
13.	req.query	An object containing a property for each query string parameter in the route.
14.	req.route	The currently-matched route, a string.
15.	req.secure	A Boolean that is true if a TLS connection is established.
16.	req.signedcookies	When using cookie-parser middleware, this property contains signed cookies sent by the request, unsigned and ready for use.
17.	req.stale	It indicates whether the request is "stale," and is the opposite of req.fresh.
18.	req.subdomains	It represents an array of subdomains in the domain name of the request.
19.	req.xhr	A Boolean value that is true if the request's "x-requested-with" header field is "xmlhttprequest", indicating that the request was issued by a client library such as jQuery

Request Object Methods

Following is a list of some generally used request object methods:

req.accepts (types)

This method is used to check whether the specified content types are acceptable, based on the request's Accept HTTP header field.

Examples:

```
req.accepts('html');
//=>?html?
req.accepts('text/html');
// => ?text/html?
```

req.get(field)

This method returns the specified HTTP request header field.

Examples:

```
req.get('Content-Type');
// => "text/plain"
req.get('content-type');
// => "text/plain"
req.get('Something');
// => undefined
```

req.is(type)

This method returns true if the incoming request's "Content-Type" HTTP header field matches the MIME type specified by the type parameter.

Examples:

```
// With Content-Type: text/html; charset=utf-8
req.is('html');
req.is('text/html');
req.is('text/*');
```

req.param(name [, defaultValue])

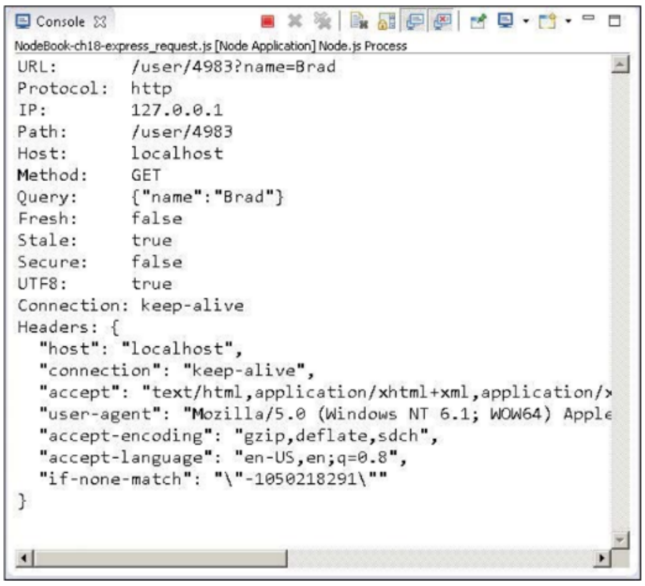
This method is used to fetch the value of param name when present.

Examples:

```
// ?name=sasha
req.param('name')
// => "sasha"
// POST name=sasha
req.param('name')
// => "sasha"
// /user/sasha for /user/:name
req.param('name')
// => "sasha"
```

Listing 18.3 `express_request.js`: **Accessing properties of the `Request` object in Express**

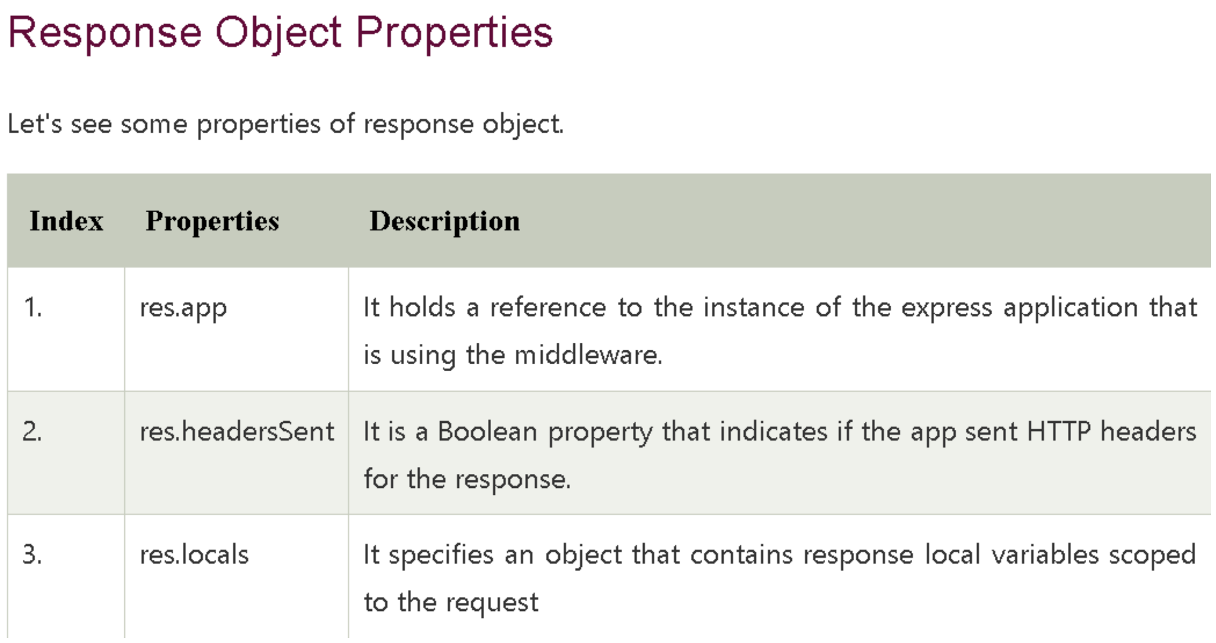
```
var express = require('express');
var app = express();
app.listen(80);
app.get('/user/:userid', function (req, res) {
  console.log("URL:\t   " + req.originalUrl);
  console.log("Protocol:  " + req.protocol);
  console.log("IP:\t   " + req.ip);
  console.log("Path:\t   " + req.path);
  console.log("Host:\t   " + req.host);
  console.log("Method:\t   " + req.method);
  console.log("Query:\t   " + JSON.stringify(req.query));
  console.log("Fresh:\t   " + req.fresh);
  console.log("Stale:\t   " + req.stale);
  console.log("Secure:\t   " + req.secure);
  console.log("UTF8:\t   " + req.acceptsCharset('utf8'));
  console.log("Connection: " + req.get('connection'));
  console.log("Headers: " + JSON.stringify(req.headers,null,2));
  res.send("User Request");
});
```


Console
NodeBook-ch18-express_request.js [Node Application] Node.js Process

```
URL:        /user/4983?name=Brad
Protocol:   http
IP:         127.0.0.1
Path:       /user/4983
Host:       localhost
Method:     GET
Query:      {"name":"Brad"}
Fresh:      false
Stale:      true
Secure:     false
UTF8:       true
Connection: keep-alive
Headers: {
  "host": "localhost",
  "connection": "keep-alive",
  "accept": "text/html,application/xhtml+xml,application/x
  "user-agent": "Mozilla/5.0 (Windows NT 6.1; WOW64) Apple
  "accept-encoding": "gzip,deflate,sdch",
  "accept-language": "en-US,en;q=0.8",
  "if-none-match": "\"-1050218291\""
}
```

<u>RESPONSE OBJECTS:</u>

- The response object (res) specifies the HTTP response which is sent by an Express app when it gets an HTTP request.
- It sends response back to the client browser.
- It facilitates you to put new cookies value and that will write to the client browser
- Once you res.send() or res.redirect() or res.sender(), you cannot do it again, otherwise, there will be uncaught error.



Response Object Properties

Let's see some properties of response object.

Index	Properties	Description
1.	res.app	It holds a reference to the instance of the express application that is using the middleware.
2.	res.headersSent	It is a Boolean property that indicates if the app sent HTTP headers for the response.
3.	res.locals	It specifies an object that contains response local variables scoped to the request

Response Object Methods

Following are some methods:

Response Append method

Syntax:

```
res.append(field [, value])
```

This method appends the specified value to the HTTP response header field. That means if the specified value is not appropriate then this method redress that.

Examples:

```
res.append('Link', ['<http://localhost/>', '<http://localhost:3000/>']);
res.append('Warning', '199 Miscellaneous warning');
```

Response Attachment method

Syntax:

```
res.attachment([filename])
```

This method facilitates you to send a file as an attachment in the HTTP response.

Examples:

```
res.attachment('path/to/js_pic.png');
```

Response Cookie method

Syntax:

```
res.cookie(name, value [, options])
```

This method is used to set a cookie name to value. The value can be a string or object converted to JSON.

Examples:

```
res.cookie('name', 'Aryan', { domain: '.xyz.com', path: '/admin', secure: true });
res.cookie('Section', { Names: [Aryan,Sushil,Priyanka] });
res.cookie('Cart', { items: [1,2,3] }, { maxAge: 900000 });
```

Response ClearCookie method

Syntax:

```
res.clearCookie(name [, options])
```

As the name specifies, the clearCookie method is used to clear the cookie specified by name.

Examples:

To set a cookie

```
res.cookie('name', 'Aryan', { path: '/admin' });
```

To clear a cookie:

```
res.clearCookie('name', { path: '/admin' });
```

Response Download method

Syntax:

```
res.download(path [, filename] [, fn])
```

This method transfers the file at path as an "attachment" and enforces the browser to prompt user for download.

Example:

```
res.download('/report-12345.pdf');
```

Response End method

Syntax:

```
res.end([data] [, encoding])
```

This method is used to end the response process.

Example:

```
res.end();
res.status(404).end();
```

Response Format method

Syntax:

```
res.format(object)
```

This method performs content negotiation on the Accept HTTP header on the request object, when present.

Example:

```
res.format({
  'text/plain': function(){
    res.send('hey');
  },
 'text/html': function(){
    res.send('
hey');
  },
  'application/json': function(){
    res.send({ message: 'hey' });
  },
 'default': function() {
    // log the request and respond with 406
    res.status(406).send('Not Acceptable');
  }
});
```

Response Get method

Syntax:

```
res.get(field)
```

This method provides HTTP response header specified by field.

Example:

```
res.get('Content-Type');
```

Response JSON method:

Syntax:

```
res.json([body])
```

This method returns the response in JSON format.

Example:

```
res.json(null)
res.json({ name: 'ajeet' })
```

Response Redirect method

Syntax:

```
res.redirect([status,] path)
```

This method redirects to the URL derived from the specified path, with specified HTTP status

Examples:

```
res.redirect('http://example.com');
```

Response Render method

Syntax:

```
res.render(view [, locals] [, callback])
```

This method renders a view and sends the rendered HTML string to the client.

Examples:

```
// send the rendered view to the client
res.render('index');
// pass a local variable to the view
res.render('user', { name: 'aryan' }, function(err, html) {
  // ...
});
```

Response Send method

Syntax:

```
res.send([body])
```

This method is used to send HTTP response.

Examples:

```
res.send(new Buffer('whoop'));
res.send({ some: 'json' });
res.send('
.....some html
');
```

Response sendFile method

Syntax:

```
res.sendFile(path [, options] [, fn])
```

This method is used to transfer the file at the given path. It sets the Content-Type response HTTP header field based on the filename's extension.

Examples:

```
res.sendFile(fileName, options, function (err) {
  // ...
});
```

Response Set method

Syntax:

```
res.set(field [, value])
```

This method is used to set the response of HTTP header field to value.

Examples:

```
res.set('Content-Type', 'text/plain');


res.set({
  'Content-Type': 'text/plain',
  'Content-Length': '123',
})
```

Response Status method

Syntax:

```
res.status(code)
```

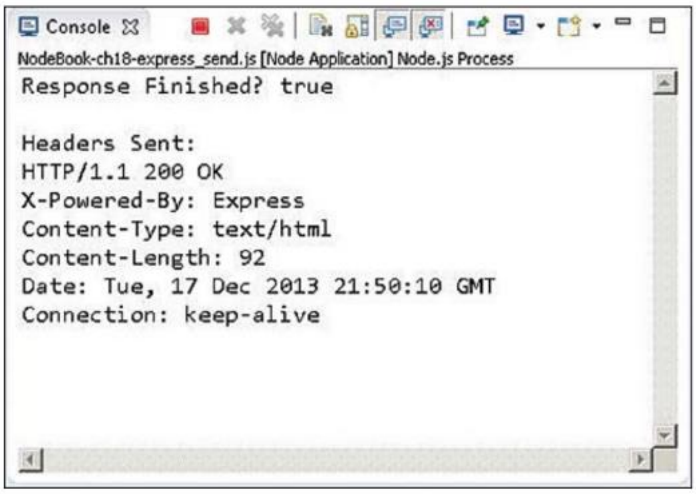
This method sets an HTTP status for the response.

Examples:

```
res.status(403).end();
res.status(400).send('Bad Request');
```

Listing 18.4 `express_send.js`: **Sending status, headers, and response data using the** `Response` **object**

```
var express = require('express');
var url = require('url');
var app = express();
app.listen(80);
app.get('/', function (req, res) {
  var response = '<html><head><title>Simple Send</title></head>' +
                 '<body><h1>Hello from Express</h1></body></html>';
  res.status(200);
  res.set({
    'Content-Type': 'text/html',
    'Content-Length': response.length
  });
  res.send(response);
  console.log('Response Finished? ' + res.finished);
  console.log('\nHeaders Sent: ');
  console.log(res.headerSent);
});
app.get('/error', function (req, res) {
  res.status(400);
  res.send("This is a bad request.");
});
```


Console
NodeBook-ch18-express_send.js [Node Application] Node.js Process

```
Response Finished? true

Headers Sent:
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html
Content-Length: 92
Date: Tue, 17 Dec 2013 21:50:10 GMT
Connection: keep-alive
```

Redirecting the Response

- A common need when implementing a webserver is the ability to redirect a request from the client to a different location on the same server or on a completely different server. The res. redirect(path) method handles redirection of the request to a new location.

Listing 18.7 `express_redirect.js`: **Redirecting requests on an Express server**

```
var express = require('express');
var url = require('url');
var app = express();
app.listen(80);
app.get('/google', function (req, res) {
  res.redirect('http://google.com');
});
app.get('/first', function (req, res) {
  res.redirect('/second');
});
app.get('/second', function (req, res) {
  res.send("Response from Second");
});
app.get('/level/A', function (req, res) {
  res.redirect("../B");
});
app.get('/level/B', function (req, res) {
  res.send("Response from Level B");
});
```

Implementing a Template Engine

Template engines provide two benefits:

■ **Simplicity**: Templates try to make it easy to generate the HTML either by a shorthand notation or by allowing JavaScript to be embedded in the HTML document directly.

■ **Speed**: Template engines optimize the process of building the HTML documents. Many compile a template and store the compiled version in a cache that makes it faster to generate the HTML response.

- Several template engines are available for use in Express.
- EJS uses special notation to embed JavaScript in normal HTML documents.
- This makes it much easier to transition from normal HTML.
- The downside is that the HTML documents are even more complex than the originals templates.
- To run the example for this section, you need to install both the Pug and EJS modules in your application using the following commands:

```
npm install Pug

npm install EJS
```

Defining the Engine:

- The first step in implementing a template engine is to define a default template engine for the Express application.
- This is done by setting the view engine setting on the express() application object.
- You also need to set the views setting to the location where your template files are stored.
- For example, the following sets the ./views directory as the root for template documents and pug as the view engine:

```
var app = express();

app.set('views', './views');

app.set('view engine', 'pug');
```

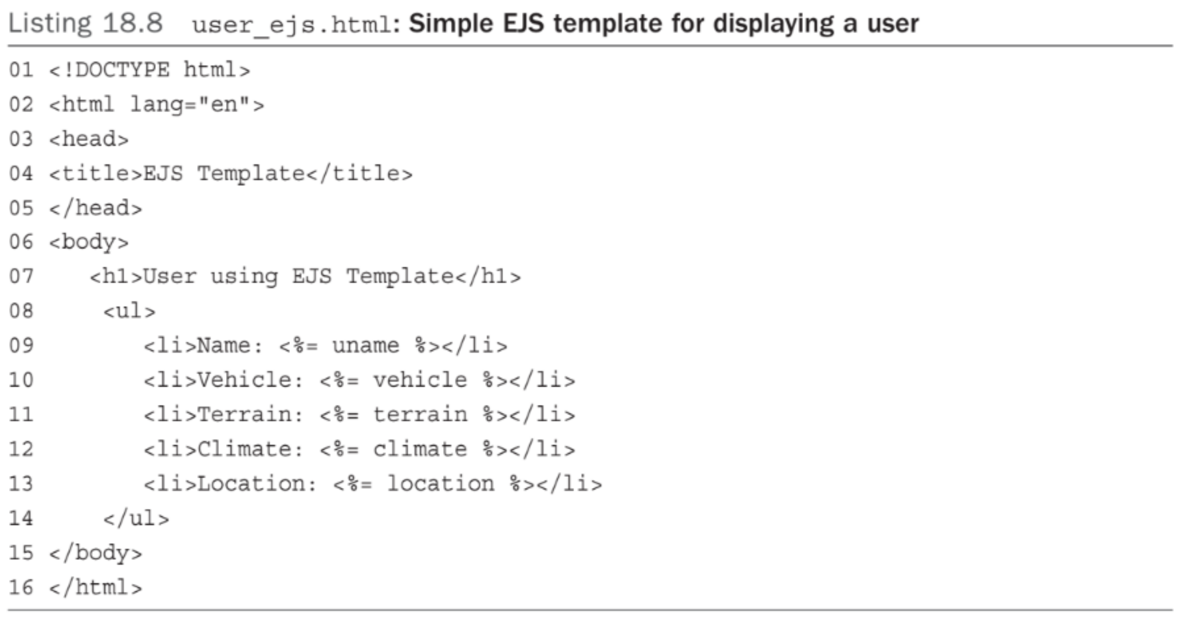
Creating Templates

You also need to create template files. When creating template files, keep in mind these considerations:

■ **Reusability**: Try to make your templates reusable in other parts of your application and in other applications. Most template engines cache the templates to speed up performance. The more templates you have requires more caching time. Try to organize your templates so that they can be used for multiple purposes. For example, if you have several tables of a data displayed in your app, only make a single template for all of them that can not only dynamically add the data, but can also set column headers, titles, and such.

■ **Size**: As template sizes grow, they tend to become more and more unwieldy. Try to keep your templates compartmentalized to the type of data they are presenting. For example, a page that has a menu bar, form, and table could be split into three separate templates.

■ **Hierarchy**: Most websites and applications are built on some sort of hierarchy. For example, the <head> section as well as a banner and menu may be the same throughout the site. Use a separate template for components that show up in multiple locations, and just include those subtemplates when building your final page.



Listing 18.8 user_ejs.html: **Simple EJS template for displaying a user**

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>EJS Template</title>
</head>
<body>
   <h1>User using EJS Template</h1>
    <ul>
       <li>Name: <%= uname %></li>
       <li>Vehicle: <%= vehicle %></li>
       <li>Terrain: <%= terrain %></li>
       <li>Climate: <%= climate %></li>
       <li>Location: <%= location %></li>
    </ul>
</body>
</html>
```

Working with Express and Template Engine:

File name:**index.js**

```
const express = require('express');
const path = require('path');
const PORT = 8085;
const app = express();

app.set('view engine','ejs');
app.set('views',path.join(__dirname,'views'));

app.use(express.urlencoded());

const studentData = [
    {name:"Satya",phone:8688220450,dept:"CSE"},
    {name:"Sravani",phone:9290760293,dept:"ECE"},
    {name:"Aparna",phone:9848016667,dept:"IT"}
];

app.get('/',function(req,res){
    return res.send('<h1>HOME PAGE<h1>');
});

app.get('/contact',function(req,res){
    res.render('ok',{stu_data : studentData});
});

app.post('/add-data',function(req,res){
    // console.log(req.body);
    studentData.push({name:req.body.fullname,phone:req.body.phone,dept:req.bod
y.dept});
    res.redirect('/contact');
});

app.listen(PORT,function(err){
    if(err){
        console.log("! SERVER IS NOT RUNNING !");
        return ;
    }
    console.log("Server is running at port : ",PORT);
});
```

File name: Ok.js (inside views folder)

```
<html>
    <head>
        <link rel="stylesheet" href="server2.css">
    </head>
    <body>
        <h1>STUDENT DATA</h1>
        <ul>
            <%for(let i of stu_data){%>
                <li>
                    <%=i.name%>
                    <%=i.phone%>
                    <%=i.dept%>
                </li>
            <%}%>
        </ul>
        <form action="add-data" method="POST">
            <input type="text" name="fullname" placeholder="Enter your name"
required>
            <input type="number" name="phone" placeholder="Enter you phone
number" required>
            <input type="text" name="dept" placeholder="Enter your department"
required>
            <button type="submit"> ADD STUDENT </button>
        </form>
    </body>
</html>
```