

Module 2

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 2 Study Guide and Deliverables

Readings:	Brown: Chapters 6–9
Assignments:	Assignment 2 due Tuesday, January 31 at 6:00 AM ET
Live Classroom:	<ul style="list-style-type: none">• Tuesday, January 24, 6:00–8:00 PM ET• Facilitator live office: TBD

■ Web Applications using Core Node.js modules

Learning Objectives

By reading the lectures and textbook, participating in the discussions, and completing the assignments, you will be able to do the following:

- Develop TCP based server and client applications.
- Write web applications using HTTP framework.
- Process request data and headers.
- Describe the basic web server flow.
- Write HTTP clients for retrieving and parsing web site data.

Introduction

The core networking modules for creating web applications using Node.js are the following:

- net – for creating TCP based server and clients
- dgram – for creating UDP/Datagram sockets
- http – for creating HTTP based web applications
- https – for creating TLS/SSL clients and servers

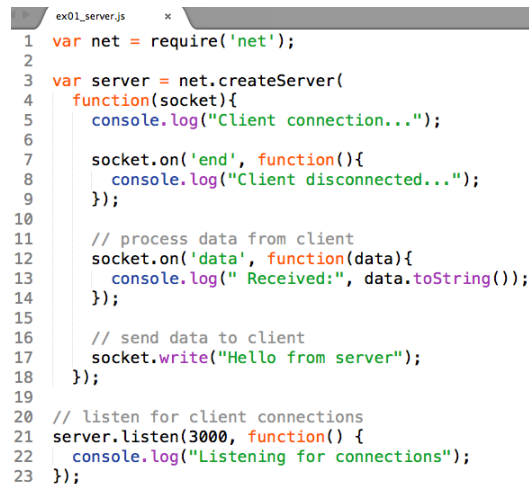
In this lecture, the net module and http module are explored in detail.

Node.js net module

The net module contains methods for creating both the server and client applications. The module's `createServer()` method is used for specifying the server functionality. The argument specified for the method is the listener for the *connection* event. When a client connects to the server, the function is invoked. The argument for the listener function is the socket object used for communicating with the client.

Once the connection is established with the client, the listener for the *data* event is fired when the server receives data from the client. Similarly, the *end* event handler is triggered when the client closes the connection. The server can communicate with the client by sending data using the `write()` method. After the server object is created, the `listen()` method is used for specifying the port through which the clients can connect to the server.

The server program is shown below. When a client connects to the server, the server immediately sends the message Hello from server to the client. When the client sends data to the server, the server prints the received data to the console. When the client closes the connection, the server prints the shown message to the console.

A screenshot of a code editor window titled 'ex01_server.js'. The code is as follows:

```
1 var net = require('net');
2
3 var server = net.createServer(
4   function(socket){
5     console.log("Client connection...");
6
7     socket.on('end', function(){
8       console.log("Client disconnected...");
9     });
10
11     // process data from client
12     socket.on('data', function(data){
13       console.log(" Received:", data.toString());
14     });
15
16     // send data to client
17     socket.write("Hello from server");
18   });
19
20 // listen for client connections
21 server.listen(3000, function() {
22   console.log("Listening for connections");
23 });
```

The net module's `connect()` method is used for specifying the client functionality. The first argument for this method is the port on which the server is listening for connections. An optional second argument can be specified for the host on which the server is running. If omitted, the value `localhost` is assumed. The last argument is the connection listener for the *connect* event. The function is invoked when the client connects to the server. In the following example, the client sends a message to server when connected.

Once the connection is established with the server, the listener for the *data* event is fired when the client receives data from the server. Similarly, the *end* event handler is triggered when the client closes the connection. The client can communicate with the server by sending data using the `write()` method.

```

ex01_client.js
1 var net = require('net');
2
3 var client = net.connect({port:3000},
4   function(){
5     console.log("Connected to server");
6     var msg = "Hello from client " +
7       Math.floor(1000*Math.random());
8     console.log("Sending: " + msg);
9     // send data to server
10    client.write(msg);
11  });
12
13 client.on('end', function(){
14   console.log("Client disconnected...");
15 });
16
17 client.on('data', function(data){
18   console.log(" Received:", data.toString());
19   client.end();
20 });

```

The following figure shows the output of the server and two client connections to the server.

The screenshot shows three terminal windows. The left window runs the server (ex01_server.js), showing it listening on port 4017 and receiving connections from clients 917 and 351. The top-right window runs the first client (ex01_client.js), showing it connecting to the server, sending a message, receiving a response, and disconnecting. The bottom-right window runs the second client (ex01_client.js), showing similar actions with client ID 351.

Case Study - Broadcast server using net module

The server program can be extended to serve as a broadcast server for multi-client communication. A client sends messages to the server. The server then broadcasts the message to all other clients.

The server program shown below keeps track of the client connects by adding the socket objects to the array named clients. When a client disconnects from the server, the corresponding socket object is removed from the clients array.

```

ex02_server.js
1 var net = require('net');
2 // Keep track of client connections
3 var clients = [];
4
5 var server = net.createServer(
6   function(socket){
7     console.log("Client connection...");
8     clients.push(socket);
9
10    socket.on('end', function(){
11     console.log("Client disconnected...");
12     // remove socket from list of clients
13     var index = clients.indexOf(socket);
14     if (index !== -1) {
15       clients.splice(index);
16     }
17   });

```

When the server receives data from a client, the same data is written to all the other clients except the one from which the message was received, as shown below.

```

ex02_server.js
18
19 socket.on('data', function(data){
20   console.log(" Received: ", data.toString());
21   // Broadcast to other clients
22   for (var i = 0; i < clients.length; i++) {
23     if (clients[i] !== socket) {
24       clients[i].write(data);
25     }
26   }
27 });
28
29 socket.write("Hello from server");
30 });
31
32 server.listen(3000, function() {
33   console.log("Listening for connections");
34 });
--

```

The client program uses the core readline module and creates the interface for reading input from the standard input and writing to the standard output. The question() method shows the prompt and waits for input from the user. The input data is sent to the server. If the user's input is bye, the client disconnects from the server. Otherwise, the client waits for the next input from the user.

```

ex02_client.js
1 var net = require('net');
2 var readline = require('readline');
3
4 var clientId = "Client " +
5   Math.floor(1000*Math.random());
6
7 var rl = readline.createInterface({
8   input: process.stdin,
9   output: process.stdout
10 });
11
12 var readMessage = function(client) {
13   rl.question("Enter Message: ", function (line){
14     client.write("From " + clientId + ": " + line);
15     if (line === "bye")
16       client.end();
17     else
18       readMessage(client);
19   });
20 };
--

```

The client program connects to the server and handles the data and end events as shown below.

```

ex02_client.js
21
22 var client = net.connect({port:3000},
23   function(){
24     console.log("Connected to server");
25     readMessage(client);
26   });
27
28 client.on('end', function(){
29   console.log("Client disconnected..");
30   return;
31 });
32
33 client.on('data', function(data){
34   console.log("\n Received:", data.toString());
35 });

```

A sample output of the server and two clients communicating with it is shown below.

```

net -- node -- 48x25
>node ex02_server.js
Listening for connections
Client connection...
Client connection...
Received: From Client 580: Message1
Received: From Client 100: Message2
Received: From Client 100: bye
Client disconnected...
Received: From Client 580: bye
Client disconnected...

net -- node -- 45x12
>node ex02_client.js
Connected to server
Enter Message:
Received: Hello from server
Message1
Enter Message:
Received: From Client 100: Message2
Message2
Received: From Client 100: bye
bye
Client disconnected...

net -- node -- 45x12
>node ex02_client.js
Connected to server
Enter Message:
Received: Hello from server
Message1
Received: From Client 580: Message1
Message2
Enter Message: bye
Client disconnected...

```

Node.js http Module

The http module contains methods for creating both the HTTP protocol server and client applications. The module's `createServer()` method is used for specifying the server functionality. The argument specified for the method is the listener for the *request* event. When a client connects to the server, the function is invoked. The arguments for the listener function are the request and response objects. The request object is a readable stream while the response object is a writable stream. The request object is examined to determine what needs to be done. The output to the client is sent through the response object. The server then listens on the specified port for incoming connections.

```

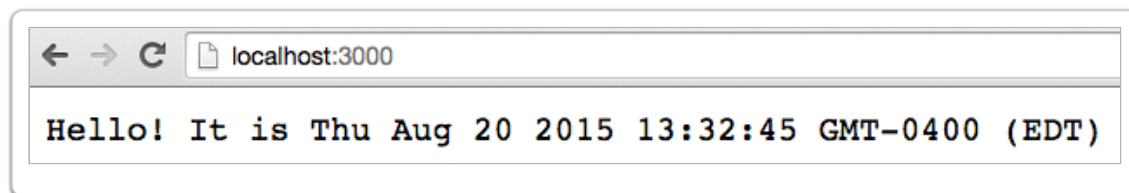
ex01_server.js
1  var http = require('http');
2
3  var server = http.createServer(
4    function (request, response) {
5      // process the request
6      console.log("Request URL:", request.url,
7        "- Request Method:", request.method);
8
9      // send the response
10     response.write('Hello! It is ' + new Date());
11     response.end();
12   });
13
14
15  server.listen(3000);
16  console.log('Server running at http://localhost:3000/');

```

Alternatively, the same program can be written using the event model as shown below.

```
1 var http = require('http');
2
3 var server = http.createServer();
4
5 server.on('request',
6   function (request, response) {
13
14   });
15
16 server.listen(3000);
17 console.log('Server running at http://localhost:3000/');
```

The typical client for connecting to a HTTP server is the web browser. The response from the server is shown below.



The following browser requests are sent to the server application.

- http://localhost:3000
- http://localhost:3000/users?name=suresh
- http://localhost:3000/users?name=suresh&id=1000

The sample output of the console logs from the server program is shown below for the above browser requests.

```
>node ex01_server.js
Server running at http://localhost:3000/
Request URL: / - Request Method: GET
Request URL: /favicon.ico - Request Method: GET
Request URL: /users?name=suresh - Request Method: GET
Request URL: /favicon.ico - Request Method: GET
Request URL: /users?name=suresh&id=1000 - Request Method: GET
Request URL: /favicon.ico - Request Method: GET
```

Server Request Object

The first argument in the callback function for each client connection is the server request object. The method property provides the type of HTTP request (GET, POST, PUT, DELETE) received from the client. The url property of the request object provides the data about the user's request. The url module's parse() method can be used to parse the GET request data into a JSON object.

```
var parsed_data =  
  require('url').parse(request.url);  
console.log(parsed_data);
```

The parsed data for the following URL is shown below.

```
Request URL: /users?name=suresh&id=1000  
{ protocol: null,  
  slashes: null,  
  auth: null,  
  host: null,  
  port: null,  
  hostname: null,  
  hash: null,  
  search: '?name=suresh&id=1000',  
  query: 'name=suresh&id=1000',  
  pathname: '/users',  
  path: '/users?name=suresh&id=1000',  
  href: '/users?name=suresh&id=1000' }
```

By default, the parse method returns the *query* property as is. The optional argument to parse the query string can be specified as shown in the following case.

```
var parsed_data =  
  require('url').parse(request.url, true);  
console.log(parsed_data);
```

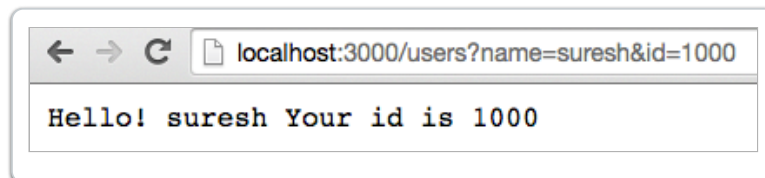
When the URL is parsed along with the query, the query parameters and their values are available in the resulting object as shown below the given URL.

```
Request URL: /users?name=suresh&id=1000  
{ protocol: null,  
  slashes: null,  
  auth: null,  
  host: null,  
  port: null,  
  hostname: null,  
  hash: null,  
  search: '?name=suresh&id=1000',  
  query: { name: 'suresh', id: '1000' },  
  pathname: '/users',  
  path: '/users?name=suresh&id=1000',  
  href: '/users?name=suresh&id=1000' }
```

The following program parses the request URL along with the query and sends the response back to the client with that information.

```
ex02_server.js
1 var http = require('http');
2
3 var server = http.createServer(
4   function (request, response) {
5     // process the request
6     console.log("Request URL:", request.url);
7     var parsed_data =
8       require('url').parse(request.url, true);
9     console.log(parsed_data);
10    // send the response
11    response.write('Hello! ' +
12      parsed_data.query.name +
13      ' Your id is ' +
14      parsed_data.query.id);
15    response.end();
16  });
17
18
19 server.listen(3000);
```

The output from the above server for the given client's request is shown below.

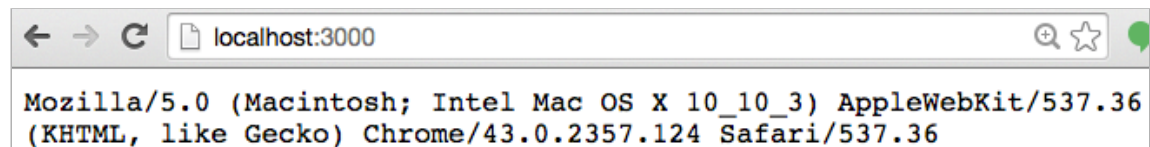


Request Headers

The headers property of the request object returns all the request headers as property-value pairs. The header values are accessed using lower-case notation of the header names. The following example responds back to the client with the value of the *user-agent* header.


```
ex03_server.js
1 var http = require('http');
2
3 var server = http.createServer(
4   function (request, response) {
5     // process the request headers
6     var headers = request.headers;
7     console.log(headers);
8     // send the response
9     response.write(headers['user-agent']);
10    response.end();
11  });
12
13
14 server.listen(3000);
```

The response to the browser shows the information about what type of browser is used to access the server application.



A screenshot of a web browser window. The address bar shows 'localhost:3000'. The main content area displays the user-agent string: 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/43.0.2357.124 Safari/537.36'.

The request headers are printed to console by the server application as shown below.

```
{ host: 'localhost:3000',
  connection: 'keep-alive',
  'user-agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/44.0.2403.155 Safari/537.36',
  accept: '*/*',
  referer: 'http://localhost:3000/',
  'accept-encoding': 'gzip, deflate, sdch',
  'accept-language': 'en-US,en;q=0.8,fr;q=0.6,id;q=0.4' }
```

Server Response Object

The server response object is used for sending the data back to the client after the incoming request is processed. The write() method can be used for sending the data. Since the response is a writable stream, the contents of a file can be read on the server side and piped to the response stream as shown in the following example.

```
.
```

The server sends the contents of the `public/home.html` file for the default GET request or for the request `/index.html`. Otherwise, an invalid access information is sent.



Case Study - Basic Web Server

The server application can be extended to provide the basic functionality of a web server for HTTP GET requests. The response object can set the MIME type of the data that is being sent after processing the request. In the following application, requests for JavaScript, PDF, JPG, GIF, text and HTML files are handled. The `mimeLookup` variable provides the mapping for the file extensions and the corresponding MIME type.



The request url property is interpreted as the file name request. If the file exists in the `public` folder relative to the application, and its MIME type can be resolved, a readable stream is created for the file and piped to the response as shown below.



The default GET request results in the following response.



The contents of the `index.html` are sent as part of the response for the above request.



If the user clicks on the first link, the contents of the file `figs.html` are sent by the server.



The contents are rendered by the browser as shown below.



When the `figs.html` file is being rendered by the browser, the requests are sent for the three image sources to the server. These requests are handled by the server and the contents of the image files are piped through the response stream for each image request.

Similarly, clicking the link for the `Nodejs.pdf` results in the contents being displayed in the browser as shown below.



Handling POST Requests

When the browser sends a POST request, the data is transmitted as the body of the request, rather than in the request url. Typical POST submission is through the HTML forms submitted to the server. The `querystring` module can be used for parsing the POST data.

The data coming as part of the POST body is captured through the data events on the server. When the entire content is received, the `end` event is emitted.

```


```

For the GET request, the contents of the following form are sent to the browser.

```


```

The form is rendered in the browser as shown below.

```


```

After the user enters the values and clicks *Submit*, the response shows the data received that is parsed and sent back in JSON format.

```


```

HTTP Clients

The HTTP client API can be used to read content from web sites. The `get()` method is used to issue GET requests to the specified URL. The callback function for this method receives the response object as its argument. The response object is a readable stream and the data is read by attaching the event handlers for the `data` and `end` events as shown below. As the data is received, the data is accumulated into a buffer. The `end` event signifies the completion of the receiving data.

```


```

The first few lines from the output of the above program is shown below.

```


```

The HTML data received from the web sites can be parsed using the third-party `cheeriojs` Node module. The module provides the `jQuery` interface to work with the HTML content.

```


```

The following program shows the content being parsed by the `cheeriojs` module. The program selects all *anchor* links in the received data and extracts the `href` attribute and the text associated with each link. The relative links are discarded and only the

absolute links are retained.

The output of the above program is shown below.

Bibliography

net Node.js Manual & Documentation, <http://nodejs.org/api/net.html>

http Node.js Manual & Documentation, <https://nodejs.org/api/http.html>

url Node.js Manual & Documentation, <https://nodejs.org/api/url.html>

Query String Node.js Manual & Documentation, <https://nodejs.org/api/querystring.html>

Cheerio Node module, <https://github.com/cheeriojs/cheerio>

Brown, E. (2014). Web Development with Node and Express. O'Reilly.

■ Web Applications using Express.js Framework

Learning Objectives

By reading the lectures and textbook, participating in the discussions, and completing the assignments, you will be able to do the following:

- Write web applications using Express framework.
- Use view engine and templates with Handlebars.
- Process request data and headers.
- Provide REST APIs for applications.
- Have cookie and session handling for applications.

Introduction

Express.js is web framework based on the Node.js *http* core module and makes writing web applications much simpler. The previous lecture using the core *http* module showed the typical code that occurs over and over again for doing related tasks such as parsing the request, parsing the cookies, managing sessions, multiple *if* statements for handling the various routes, extracting URL parameters, handling errors, etc. Express.js provides ways to reuse code and provides an MVC-like structure for building web applications.

Express.js is a third-party Node module and installed for the application as shown below.



Basic Express Web Application

The *express* module is imported into the web application and the express application (*app*) is created using the `express()` function. The `get()` method of the express *app* is used for defining the routes and the corresponding handlers. The method handles the HTTP GET requests. The first argument is the URL path (a string or a regular expression) that the route corresponds to.



The examples in the lecture require each program to be executed individually, as all the programs create a web server using the port 3000.



With the web server running, the web application is accessed using the browser as shown below. The default GET request is the route specified by the path `/`. The output to browser is shown in the figure below. However, accessing any other path returns the message that it cannot get that path.



The following example specifies the route for the default GET request (`/`) and also for the GET request with the path `/about`. In the first case, plain text is sent for the response, while in the second case, HTML data is sent. The 404 handler is also specified when the browser sends a request that doesn't match the specified routes in the web application. The `use()` method is used for specifying the 404 handler.



After executing the above program, the web application is accessed from the browser and tested with the three scenarios (`/`, `/about`, and `/foo`) as shown below.



Express provides modified methods for the *response* object to be used instead of the Node's low-level methods. The `send()` method is used for sending the content using the response object.

In addition to the `get()` method for handling HTTP GET requests, the `post()` method handles the HTTP POST requests, the `put()` method handles the HTTP PUT requests, and the `delete()` method handles the HTTP DELETE requests. The `all()` method can also be used to match all HTTP request types.

Views and Layouts - Express Handlebars

In the previous examples, the response to be set back is written in the JavaScript code using the methods of the *response* object. For simple responses, this approach is sufficient, but becomes unmanageable for complex responses. These could be multiple lines of HTML with formatting, or navigating through the data and generating the HTML. *Handlebars* is the Express template engine which allows the user to write the views in standard HTML pages and at the same time provide syntax for inserting dynamic content into the pages. In addition to the view templates, layout pages can also be specified for common look and feel that all the view pages can inherit.

The Express Handlebars module is installed using *npm* as shown below.

```
npm install express-handlebars
```

For the Express web applications, the Handlebars template engine is specified using the *engine()* method. The template engine is initialized using the Handlebars module constructor function. The argument is the *defaultLayout* page that will be used for all the views. In the following example, the template engine is set to use handlebars as the extension for all the view and layout pages.

```
const app = express();
const hbs = handlebars({
  defaultLayout: 'main.handlebars'
});
app.engine('html', hbs.compile);
app.set('view engine', 'html');
app.set('views', path.resolve(__dirname, 'views'));
app.use(express.static(path.resolve(__dirname, 'public')));
app.get('/', (req, res) => {
  res.render('home.handlebars');
});
```

The default layout view page for the example will be the *main.handlebars* page. All view pages will exhibit this common layout. The content from the appropriate view page will be rendered in the placeholder `{{body}}`. The triple curly brackets in the template and layout pages render the HTML tags, where the double curly brackets escape the HTML tags and show the content as is without rendering.

```
const app = express();
const hbs = handlebars({
  defaultLayout: 'main.handlebars'
});
app.engine('html', hbs.compile);
app.set('view engine', 'html');
app.set('views', path.resolve(__dirname, 'views'));
app.use(express.static(path.resolve(__dirname, 'public')));
app.get('/', (req, res) => {
  res.render('home.handlebars');
});
```

The Express application provides the logic for rendering the views for the corresponding router request paths. The *render* method of the *response* object specifies the view that should be shown to the user. The view pages will be rendered in the context of the default layout page.

```
const app = express();
const hbs = handlebars({
  defaultLayout: 'main.handlebars'
});
app.engine('html', hbs.compile);
app.set('view engine', 'html');
app.set('views', path.resolve(__dirname, 'views'));
app.use(express.static(path.resolve(__dirname, 'public')));
app.get('/', (req, res) => {
  res.render('home.handlebars');
});
```

The three views used in the above example are the *home*, *about*, and *404*. The respective view pages with the *handlebars* extension are shown below.

```
const app = express();
const hbs = handlebars({
  defaultLayout: 'main.handlebars'
});
app.engine('html', hbs.compile);
app.set('view engine', 'html');
app.set('views', path.resolve(__dirname, 'views'));
app.use(express.static(path.resolve(__dirname, 'public')));
app.get('/', (req, res) => {
  res.render('home.handlebars');
});
```

The view pages and the layout pages have the following structure within the application.

```
const app = express();
const hbs = handlebars({
  defaultLayout: 'main.handlebars'
});
app.engine('html', hbs.compile);
app.set('view engine', 'html');
app.set('views', path.resolve(__dirname, 'views'));
app.use(express.static(path.resolve(__dirname, 'public')));
app.get('/', (req, res) => {
  res.render('home.handlebars');
});
```

The web application is run as shown below.

```
const app = express();
const hbs = handlebars({
  defaultLayout: 'main.handlebars'
});
app.engine('html', hbs.compile);
app.set('view engine', 'html');
app.set('views', path.resolve(__dirname, 'views'));
app.use(express.static(path.resolve(__dirname, 'public')));
app.get('/', (req, res) => {
  res.render('home.handlebars');
});
```

The routes corresponding to the application can now be tested with the browser. The default GET request maps to the `/` route. The *home.handlebars* view is shown to the user.

```
const app = express();
const hbs = handlebars({
  defaultLayout: 'main.handlebars'
});
app.engine('html', hbs.compile);
app.set('view engine', 'html');
app.set('views', path.resolve(__dirname, 'views'));
app.use(express.static(path.resolve(__dirname, 'public')));
app.get('/', (req, res) => {
  res.render('home.handlebars');
});
```

For the GET request with the `/aboutroute`, the contents of the view `about.handlebars` is shown.

For any other request, the 404 handler triggers the view shown in the `404.handlebars` page.

Boston University Metropolitan College