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
Discussions:	Discussion 2 Due: Tuesday, July 17 at 6:00 AM ET
Assignments:	Assignment 2 Due: Tuesday, July 17 at 6:00 AM ET
Live Classrooms:	Wednesday, July 11 from 8:00-10:00 PM ET

Web Applications using Core Node.js modules

Objectives

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

- Develop TCP based server and client applications
- Write web applications using HTTP framework
- Process request data and headers
- Understand 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 specified 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.

ex01_server.js

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

var server = net.createServer(
  function(socket){
    console.log("Client connection...");

    socket.on('end', function(){
      console.log("Client disconnected...");
    });

    // process data from client
    socket.on('data', function(data){
      console.log(" Received:", data.toString());
    });

    // send data to client
    socket.write("Hello from server");
  });

// listen for client connections
server.listen(3000, function() {
  console.log("Listening for connections");
});
```

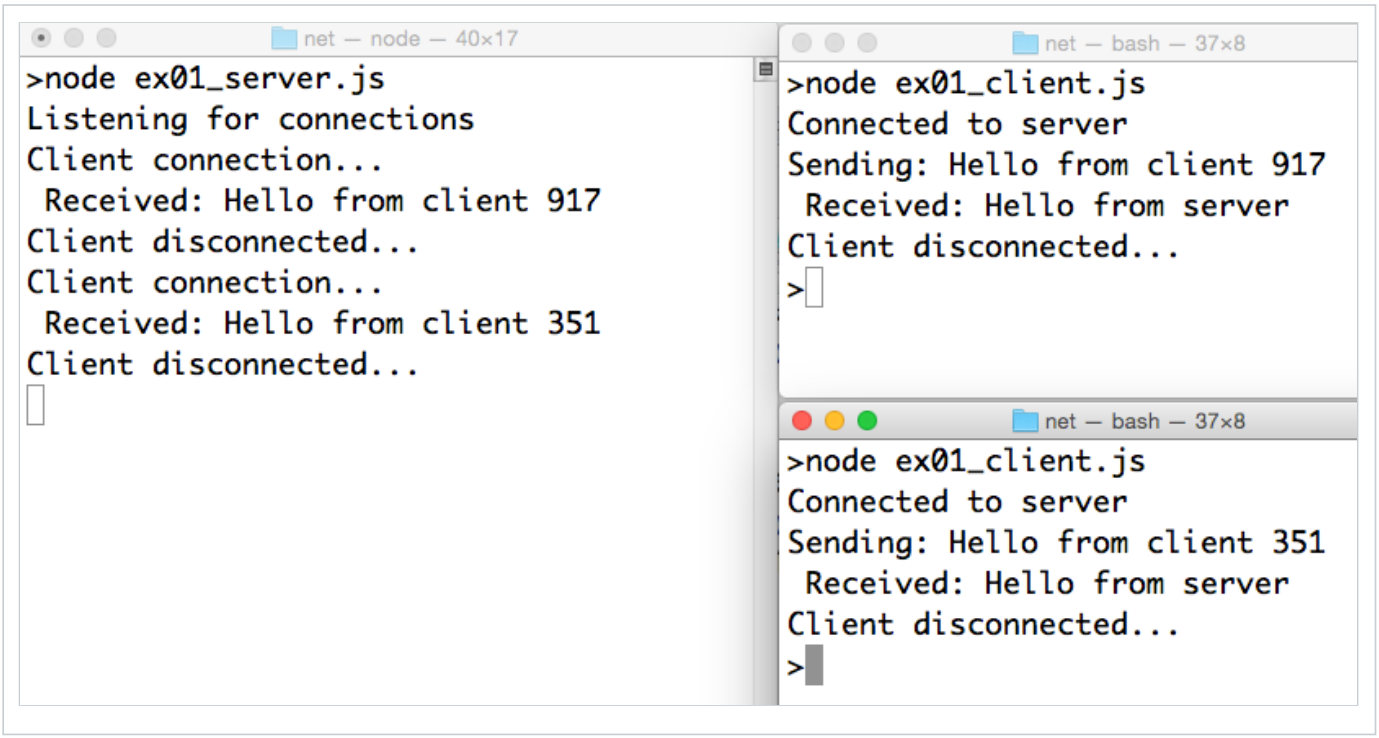
The net module's connect() method is used for specifying the client functionality. The first argument for this method is the port on which the server the listening for connections. An optional second argument can be specified for the host on which the server is running. If omitted, the value localhost 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

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

var client = net.connect({port:3000},
  function(){
    console.log("Connected to server");
    var msg = "Hello from client " +
              Math.floor(1000*Math.random());
      console.log("Sending: " + msg);
      // send data to server
      client.write(msg);
  });

client.on('end', function(){
  console.log("Client disconnected...");
});

client.on('data', function(data){
  console.log(" Received:", data.toString());
  client.end();
});
```

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



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

```
var net = require('net');
// Keep track of client connections
var clients = [];

var server = net.createServer(
  function(socket){
    console.log("Client connection...");
    clients.push(socket);

    socket.on('end', function(){
      console.log("Client disconnected...");
      // remove socket from list of clients
      var index = clients.indexOf(socket);
      if (index != -1) {
        clients.splice(index);
      }
    });
```

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

ex02_server.js

```

    socket.on('data', function(data){
      console.log(" Received: ", data.toString());
      // Broadcast to other clients
      for (var i = 0; i < clients.length; i++) {
        if (clients[i] != socket) {
          clients[i].write(data);
        }
      }
    });

    socket.write("Hello from server");
  });

server.listen(3000, function() {
  console.log("Listening for connections");
});
```

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

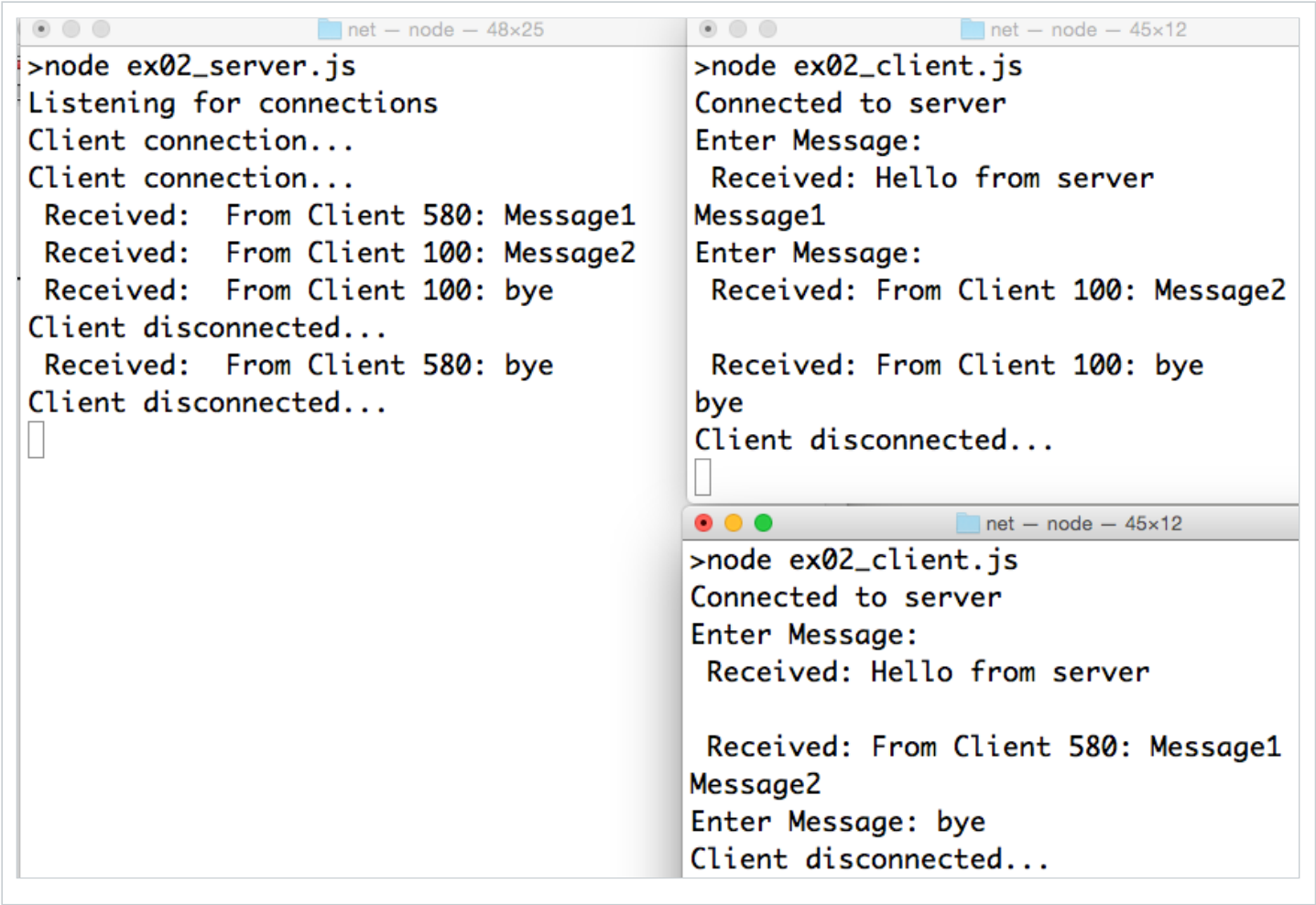
```
var net = require('net');
var readline = require('readline');

var clientId = "Client " +
      Math.floor(1000*Math.random());

var rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

var readMessage = function(client) {
  rl.question("Enter Message: ", function (line){
      client.write("From " + clientId + ": " + line);
      if (line == "bye")
        client.end();
      else
        readMessage(client);
  });
};
```

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

ex02_client.js

```

var client = net.connect({port:3000},
  function(){
    console.log("Connected to server");
    readMessage(client);
  });

client.on('end', function(){
  console.log("Client disconnected...");
  return;
});

client.on('data', function(data){
  console.log("\n Received:", data.toString());
});
```

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



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.

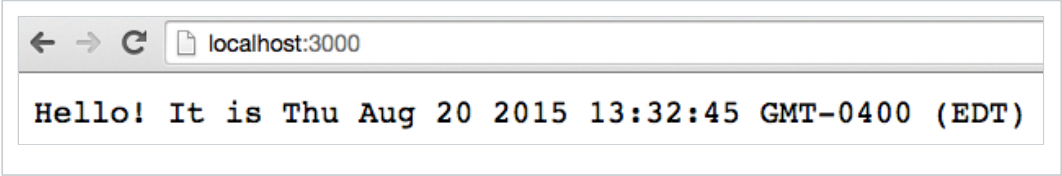
ex01_server.js

```
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

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

var server = http.createServer(
  function (request, response) {
    // process the request
    console.log("Request URL:", request.url);
    var parsed_data =
      require('url').parse(request.url, true);
    console.log(parsed_data);
    // send the response
    response.write('Hello! ' +
      parsed_data.query.name +
      ' Your id is ' +
      parsed_data.query.id);
    response.end();

  });

server.listen(3000);
```

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

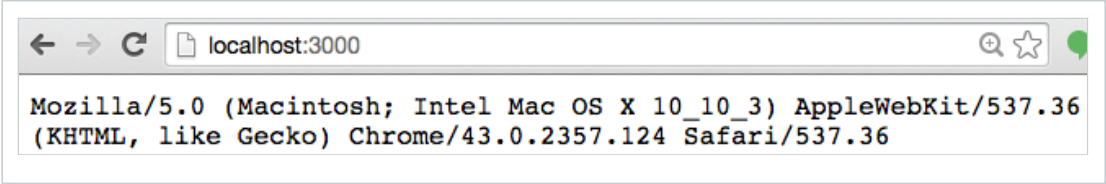
Request Headers

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

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

var server = http.createServer(
  function (request, response) {
    // process the request headers
    var headers = request.headers;
    console.log(headers);
    // send the response
    response.write(headers['user-agent']);
    response.end();

  });

server.listen(3000);
```

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



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.

ex04_server.js

```
1  var http = require('http');
2  var fs   = require('fs');
3
4  var server = http.createServer(
5      function (request, response) {
6      // process the request
7      console.log("Request URL:", request.url);
8      if ((request.method == 'GET') &&
9              ((request.url == '/') ||
10                 (request.url == '/index.html'))) {
11         var fileName = './public/home.html';
12         var rs = fs.createReadStream(fileName);
13         // send the response
14         rs.pipe(response);
15     } else {
16         response.write('Invalid access ' + request.url);
17         response.end();
18     }
19     });
20
21 server.listen(3000);
22 console.log('Server running at http://localhost:3000/');
```

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.

ex05_server.js

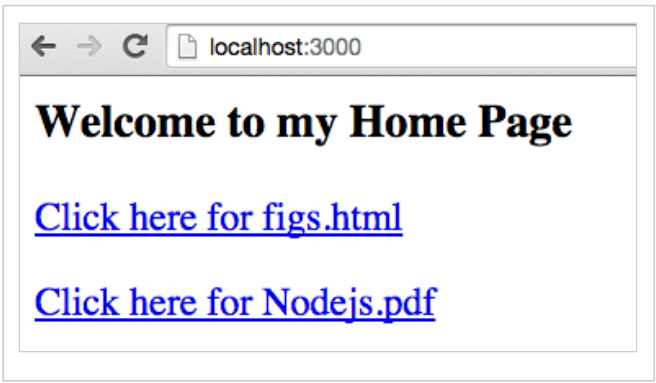
```
var http = require('http');
var fs   = require('fs');
var path = require('path');

var mimeLookup = {
  '.js'  : 'application/javascript',
  '.pdf' : 'application/pdf',
  '.jpg' : 'application/jpeg',
  '.gif' : 'application/gif',
  '.txt' : 'text/plain',
  '.html': 'text/html'
};

function sendError(text, response) {
  response.writeHead(404,
      { 'Content-Type': 'text/plain' });
  response.write('Error 404: ' + text);
  response.end();
}
```

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.

ex05_server.js

```
var server = http.createServer(
  function (request, response) {
    // process the request
    console.log("Request URL:", request.url);
    if (request.method == 'GET') {
      var fileName;
      if (request.url == '/') {
        fileName = '/index.html';
      } else {
        fileName = request.url;
        fileName = fileName.replace(/\.\./g, "_");
      }
      var filepath = path.resolve('./public' + fileName);
      console.log(filepath);
      // determine mime type
      var fileExt = path.extname(filepath);
      var mimeType = mimeLookup[fileExt];
      if (!mimeType) {
        sendError('Unknown MIME type', response);
        return;
      }
      // check if file exists
      fs.exists(filepath, function (exists) {
        // if not
        if (!exists) {
          sendError('Resource not found', response);
          return;
        };
        // send the file
        response.writeHead(200, { 'content-type': mimeType });
        fs.createReadStream(filepath).pipe(response);
      });
    }
  });
```

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.

index.html

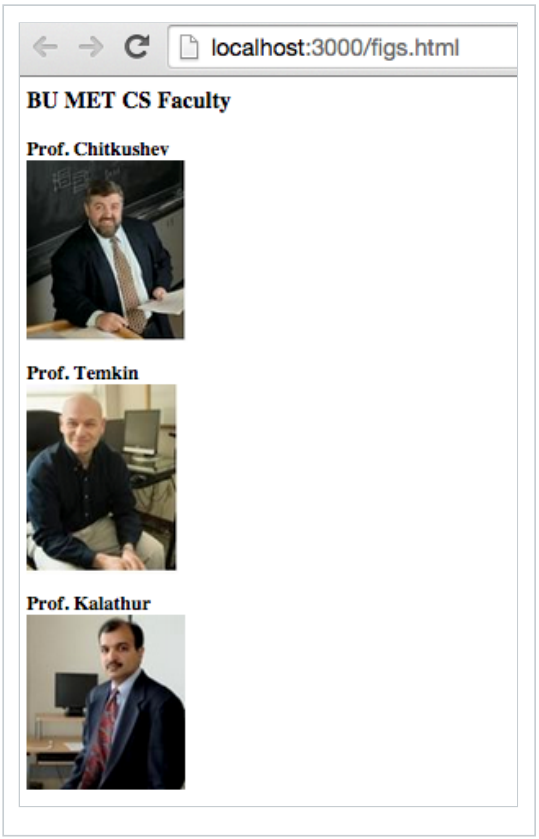
```
<html>
<body>
<h3>Welcome to my Home Page</h3>
<p>
<a href="figs.html" target=_fig>Click here for figs.html</a>
<p>
<a href="Nodejs.pdf" target=_fig>Click here for Nodejs.pdf</a>
</body>
</html>
```

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

figs.html

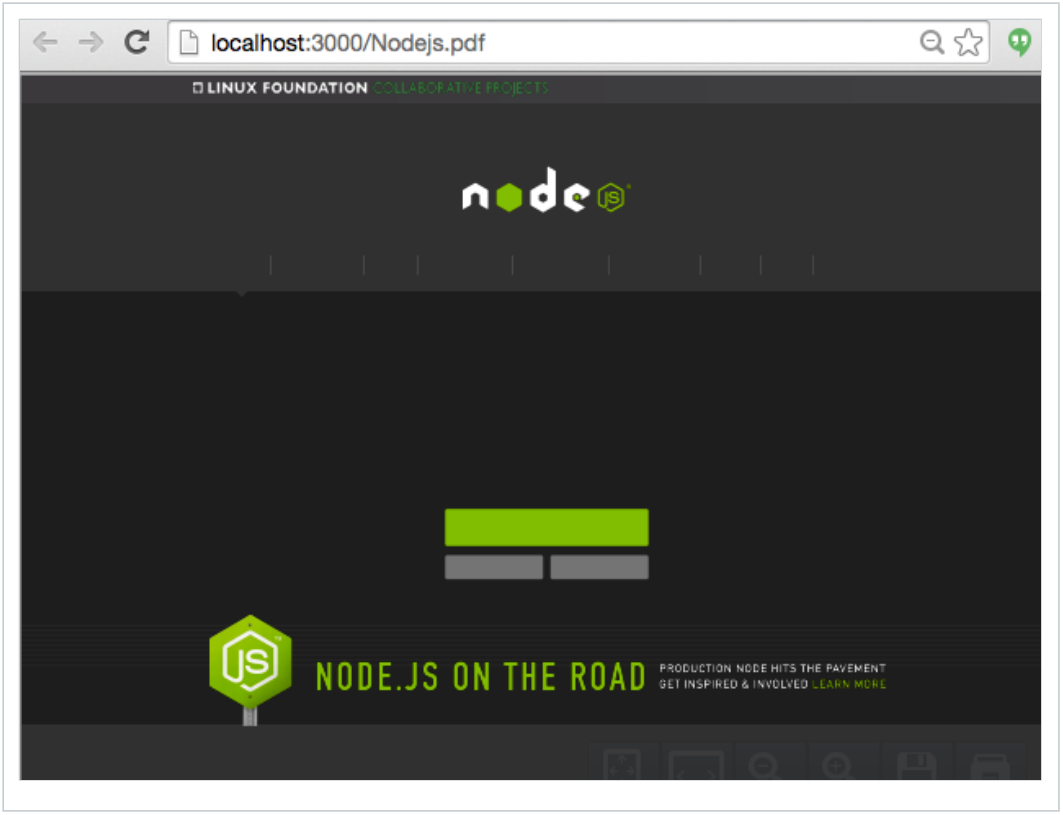
```
<html>
<body>
<h3>BU MET CS Faculty</h3>

<b>Prof. Chitkushev</b>
<br>
<img src="fac1.jpg"></img>
<p>

<b>Prof. Temkin</b>
<br>
<img src="fac2.jpg"></img>
<p>

<b>Prof. Kalathur</b>
<br>
<img src="fac3.jpg"></img>
<p>
</body>
</html>
```

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.

ex06_server.js

```
var http = require('http');
var fs   = require('fs');
var qs   = require('querystring');

var server = http.createServer(
  function (request, response) {
    // process the request
    console.log("Request URL:", request.url);
    if (request.method == 'POST')  {
      var data = '';

      request.on('data', function(chunk){
        data += chunk;
      });

      request.on('end', function(){
        var postData = qs.parse(data);
        //send the response
        response.writeHead(200,
          { 'content-type': 'application/json' });
        response.write(JSON.stringify(postData));
      });
    } else {
      // send the form
      var fileName = './public/form.html';
      var rs = fs.createReadStream(fileName);
      rs.pipe(response);
    }
  });

server.listen(3000);
console.log('Server running at http://localhost:3000/');
```

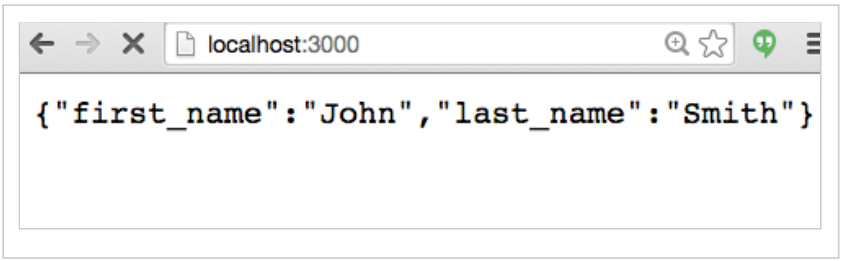
For the GET request, the contents of the following form are sent to the browser.

form.html

```
<html>
<body>
  <h3>Post Submission</h3>
  <form method="post" action="/">
    <table>
      <tbody>
       <tr>
        <th width="200">First Name: </th>
        <td><input type="text" name="first_name" required
              placeholder="Enter first name"></td>
       </tr>
       <tr>
         <th>Last Name: </th>
         <td><input type="text" name="last_name" required
              placeholder="Enter last name"></td>
       </tr>
      </tbody>
      <tfooter>
        <tr>
          <td></td>
          <td>
           <input type="submit" value="Submit">
          </td>
        </tr>
      </tfooter>
    </table>
  </form>
</body>
</html>
```

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.

ex07_client.js

```
1  var http = require('http');
2
3  var url =
4    "http://people.bu.edu/kalathur" +
5    "/current_courses.html";
6
7  var req = http.get(url, function (response){
8    var buffer = '';
9
10   response.on('data', function(chunk){
11     buffer += chunk;
12   });
13
14   response.on('end', function(){
15     console.log(buffer);
16   });
17 });
18
19 req.on('error', function (err){
20   console.log(err);
21 })
```

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

```
>node ex07_client.js
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html><!-- InstanceBegin template="/Templates/master.dwt" codeO
eHTMLIsLocked="false" -->
<!-- DW6 -->
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso
-1">
<!-- InstanceBeginEditable name="doctitle" -->
<title>Suresh Kalathur</title>
```

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.

```
>npm install --save cheeriojs
```

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.

ex08_client.js

```
1  var http = require('http');
2  var cheerio = require('cheerio');
3
4  var url =
5    "http://people.bu.edu/kalathur" +
6    "/current_courses.html";
7
8  var req = http.get(url, function (response){
9    var data = '';
10
11   response.on('data', function(chunk){
12     data += chunk;
13   });
14
15   response.on('end', function(){
16     var $ = cheerio.load(data);
17     var linkURLs = [];
18     $('a').map(function(index, elem) {
19       var href = $(elem).attr('href');
20       var text = $(elem).html().trim();
21       if (href.indexOf('http:') == 0) {
22         linkURLs.push({'link': href,
23                        'data': text});
24       }
25     });
26     console.log(linkURLs);
27   });
28 });
29
30 req.on('error', function (err){
31   console.log(err);
32 })
```

The output of the above program is shown below.

```
>node ex08_client.js
[ { link: 'http://www.bu.edu/csmet', data: 'My Department' },
  { link: 'http://www.bu.edu/met', data: 'Metropolitan College' },
  { link: 'http://kalathur.com/courses/?course_id=cs701_15_summer',
    data: 'CS701 OL - Rich Internet Application Development (Summer2,
 Online Course)' },
  { link: 'http://kalathur.com/courses/?course_id=cs520ol_15_summer',
    data: 'CS520 OL - Information Structures (Summer2, Online Course)
' } ]
```

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

Ethan Brown, Web Development with Node and Express, O'Reilly, 2014.

Web Applications using Express.js Framework

Objectives

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

- 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.

```
>npm install --save express
```

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.

ex01_express.js

```
1  var express = require('express');
2  var app = express();
3
4  // GET request to the homepage
5  app.get('/', function(req, res) {
6    res.type('text/plain');
7    res.send('Welcome to CS602!');
8  });
9
10 app.listen(3000, function(){
11   console.log('http://localhost:3000');
12 });
```

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

```
>node ex01_express.js
http://localhost:3000
```

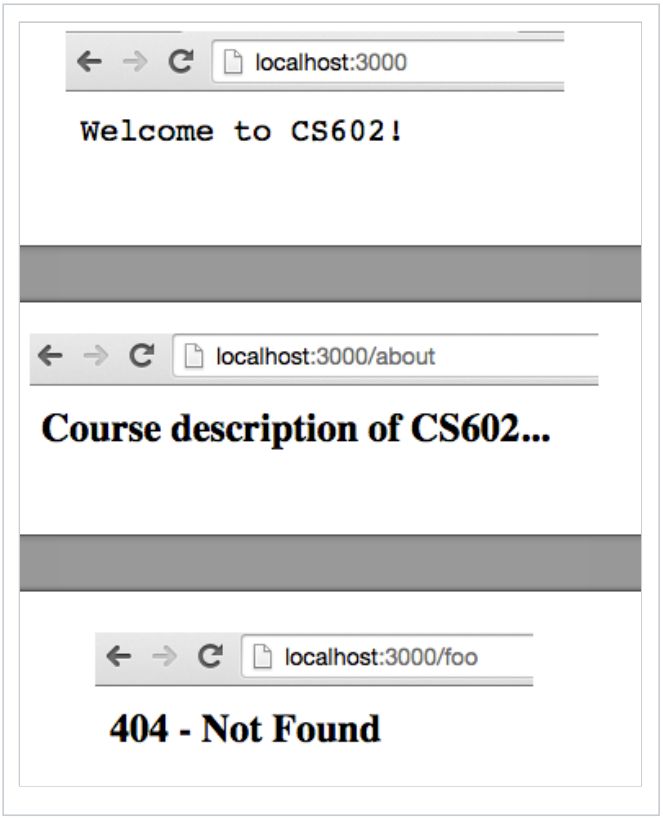
With the web server running, the web application is accessed using the browser as shown below. The default GET request is the route specified by the path /. The output to browser is shown in the figure below. However, accessing any other path returns the message that it cannot get that path.

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.

ex02_express.js

```
1  var express = require('express');
2  var app = express();
3
4  // GET request to the homepage
5  app.get('/', function(req, res) {
6    res.type('text/plain');
7    res.send('Welcome to CS602!');
8  });
9
10 app.get('/about', function(req, res) {
11   res.type('text/html');
12   res.send('<b>Course description of CS602...</b>');
13 });
14
15 app.use(function(req, res) {
16   res.type('text/html');
17   res.status(404);
18   res.send("<b>404 - Not Found</b>");
19 });
20
21 app.listen(3000, function(){
22   console.log('http://localhost:3000');
23 });
```

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 --save 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.

ex03_express.js

```
1  var express = require('express');
2  var app = express();
3
4  // setup handlebars view engine
5  var handlebars = require('express-handlebars');
6
7  app.engine('handlebars',
8    handlebars({defaultLayout: 'main'}));
9
10 app.set('view engine', '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.

main.handlebars

```
<!doctype html>
<html>
<head>
  <title>CS602</title>
</head>
<body>
<h2>Welcome to CS602!</h2>
    <a href="/">Home</a>|
    <a href="/about">About</a> |
    <a href="/foo">Foo</a> |
<hr/>

  {{{body}}}

</body>
</html>
```

The Express application provides the logic for rending 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.

ex03_express.js

```
// GET request to the homepage
app.get('/', function(req, res) {
  res.render('home');
});

app.get('/about', function(req, res) {
  res.render('about');
});

app.use(function(req, res) {
  res.status(404);
  res.render('404');
});

app.listen(3000, function(){
  console.log('http://localhost:3000');
});
```

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.

home.handlebars

```
1  <h3>Home Page for CS602!</h3>
```

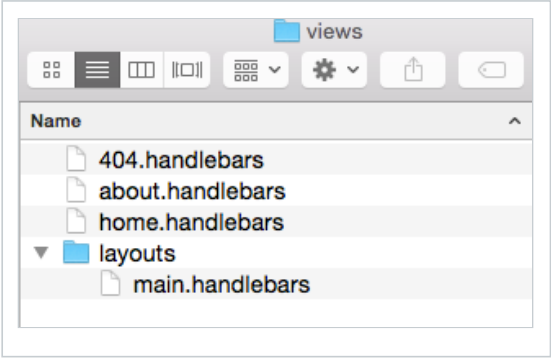
about.handlebars

```
1  <h3>About CS602...</h3>
```

404.handlebars

```
1  <h3>404 – Not Found</h3>
```

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



The web application is run as shown below.

```
>node ex03_express.js
http://localhost:3000
```

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.image 7

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.

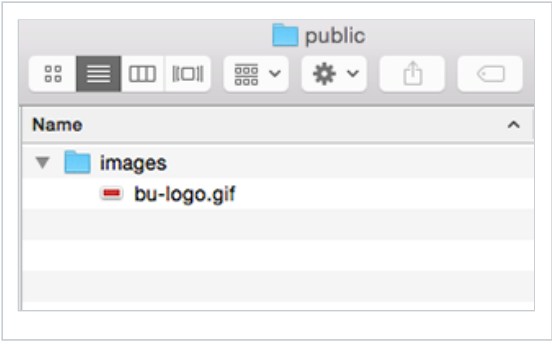
Static content

For serving static content (image files, CSS files, client-side JavaScript files, etc.), it is customary in Express applications to keep all the content under the *public* folder of the application. Before any routes are specified, the static() method is used to specify the location from which the resources will be delivered.

ex04_express.js

```
1  var express = require('express');
2  var app = express();
3
4  // setup handlebars view engine
5  var handlebars = require('express-handlebars');
6
7  app.engine('handlebars',
8    handlebars({defaultLayout: 'main_logo'}));
9
10 app.set('view engine', 'handlebars');
11
12 // static resources
13 app.use(express.static(__dirname + '/public'));
```

In the following example, an image file is placed under the *public/images* folder of the application.

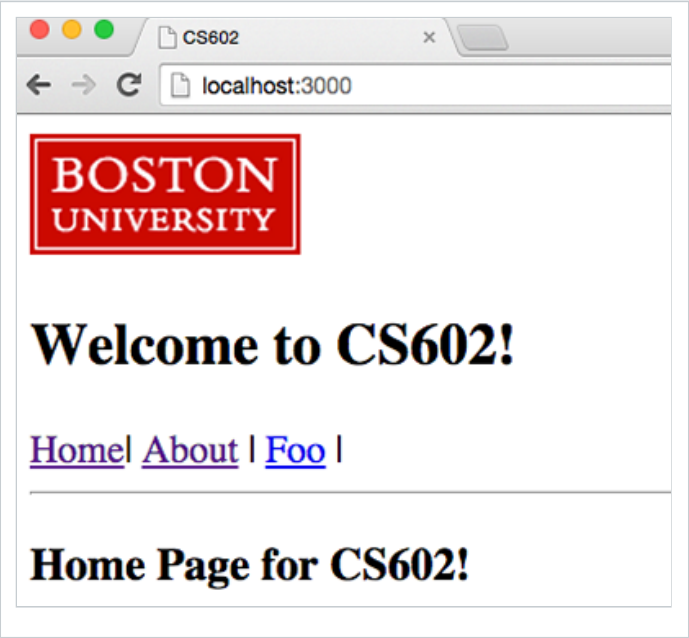


Within the view pages, the static resources are referred without the */public* prefix as shown in Line#8 below.

main_logo.handlebars

```
1  <!doctype html>
2  <html>
3  <head>
4    <title>CS602</title>
5  </head>
6  <body>
7  <header>
8    <img src="/images/bu-logo.gif" alt="logo">
9  </header>
10 <h2>Welcome to CS602!</h2>
11     <a href="/">Home</a>|
12     <a href="/about">About</a> |
13     <a href="/foo">Foo</a> |
14 <hr/>
15
16   {{{body}}}
17
18 </body>
19 </html>
```

Accessing the home page of the application renders the logo from the layout as shown below.



Accessing Model Data in View Templates

The view templates so far had only static content. When the request is handled by the Express application, any data that needs to shown in the view template can be specified through the render() method of the *response.* In the following example, a module is exported with two functions, *getRandomQuote* and *getAllQuotes*. The data for these methods is an array of quotes by famous people.

ex04_quotes.js

```
1  // http://www.forbes.com/sites/kevinkruse/2013/05
2
3  var quotes = [
4    "Life isn't about getting and having, it's abou
5    "Whatever the mind of man can conceive and beli
6    "Strive not to be a success, but rather to be o
7    "You miss 100% of the shots you don't take. -Wa
8    "Every strike brings me closer to the next home
9    "Definiteness of purpose is the starting point
10   "Life is what happens to you while you're busy
11   "We become what we think about. -Earl Nightinga
12   "Life is 10% what happens to me and 90% of how
13   ];
14
15
16 exports.getRandomQuote = function() {
17   return quotes[Math.floor(
18                 Math.random() * quotes.length)];
19 };
20
21 exports.getAllQuotes = function() {
22   return quotes;
23 }
```

The above module is used in the web application. When the request is for the /about GET method, the *about_quote* view template is rendered. The template is also provided with the context data as a JSON object with two properties, *quote* and *quotes*, the data for which is obtained from

the imported module.

ex04_express.js

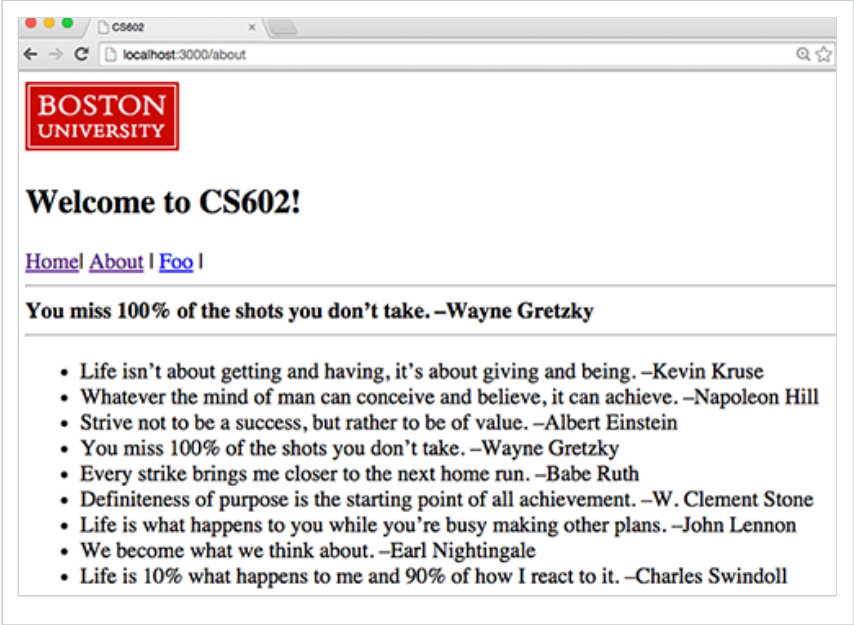
```
15  // Use the quotes module
16  var quotes = require('./ex04_quotes.js');
17
18  // GET request to the homepage
19  app.get('/', function(req, res) {
20    res.render('home');
21  });
22
23  app.get('/about', function(req, res) {
24    res.render('about_quote',
25      {quote: quotes.getRandomQuote(),
26       quotes: quotes.getAllQuotes()});
27  });
28
29  app.use(function(req, res) {
30    res.status(404);
31    res.render('404');
32  });
33
34  app.listen(3000, function(){
35    console.log('http://localhost:3000');
36  });
```

The view template accesses the context data as shown below. The randomly selected quote is first displayed. Since the *quotes* property is an array, the each helper is used to iterate over the collection and render each element. Each item in the collection is referred to as this. In addition to the item, if the iteration is over an object, @key is used for accessing the name of the property, or in the case of an array, @index is used for the index of the array.

about_quote.handlebars

```
1  <b>{{quote}}</b>
2  <hr/>
3  <ul>
4      {{#each quotes}}
5        <li>{{this}}</li>
6      {{/each}}
7  </ul>
```

The view template is rendered as shown below.



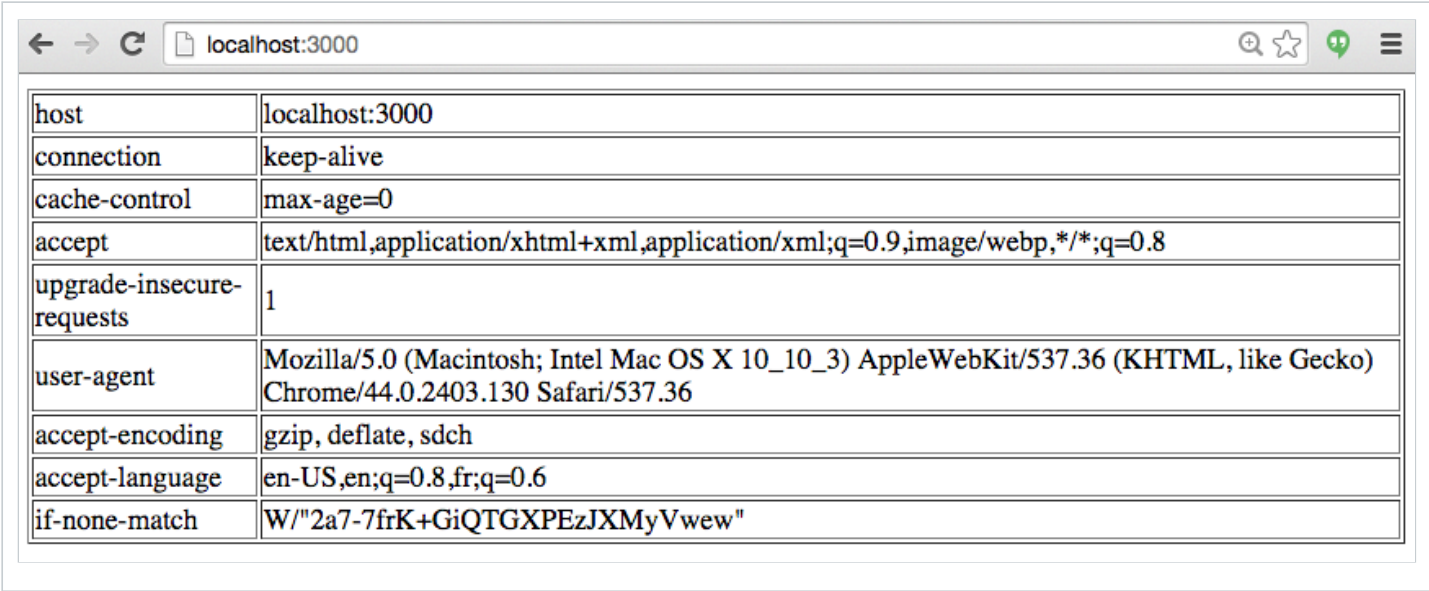
Request Headers

The HTTP request message headers can be accessed through the *headers* property of the request object. The *headers* data is an array of JSON objects, with the key of each object being the header name and the value being the associated value for the header. The following example displays all the headers received as part of the response.

ex05_headers.js

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

// GET request to the homepage
app.get('/', function(req, res) {
  res.type('text/html');
  var result = '<table border=1>';
  var item = '';
  for (var header in req.headers) {
    item = '<tr><td>' + header + '</td>' +
           '<td>' + req.headers[header] + '</td></tr>\n';
    result += item;
  }
  result += '</table>'
  res.send(result);
});

app.listen(3000, function(){
  console.log('http://localhost:3000');
});
```

The output from the above program is shown below.



A commonly used header is the *user-agent* identifying the type of the browser that the user is using for accessing the web application. In the following example, based on the *user-agent* field, the browser is redirected to the website of the browser's provider.

ex06_headers.js

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

// GET request to the homepage
app.get('/', function(req, res) {

  var userAgent = req.headers['user-agent'];
  // redirect based on browser
  if (userAgent.indexOf("Firefox") > 0)
    res.redirect("http://www.mozilla.org");
  else if (userAgent.indexOf("Chrome") > 0)
    res.redirect("http://www.google.com");
  else
    res.redirect("http://www.microsoft.com");
});

app.listen(3000, function(){
  console.log('http://localhost:3000');
});
```

Response Headers

In addition to the standard headers that go back to the browser as part of the response, the application can insert additional header information using the set() method of the Express enhanced *response* object. In the following example, the *Refresh* header is set instructing the browser to issue an automatic refresh in 5 seconds.

ex07_headers.js

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

// GET request to the homepage
app.get('/', function(req, res) {
  res.set('Refresh', 5);
  res.send(new Date().toString());
});

app.listen(3000, function(){
  console.log('http://localhost:3000');
});
```

The above technique can be used when computations as part of the request take longer time and the user is shown partial results, as the computations are ongoing.

Accessing Request Data

Data from the clients is passed to the web application through the HTTP requests (GET, PUT, POST, or DELETE). There are three ways in which data can be retrieved in the web application. If the data is passed through the request query with a question mark followed by *name=value* pairs, the *request* object's *query* property provides the data as key-value pairs. If the data is passed in the request path, the *request* object's *params* property provides the data as key-value pairs. If the data is passed through a form submission using the POST method, the *request* object's *body* property provides the data as key-value pairs. However, the following module is required for the web application to parse the data from the request's body.

```
>npm install --save body-parser
```

The following sample code shows the different types of requests and how the data is accessed. The *bodyParser's* options are set to parse the *urlencoded* bodies as well JSON data.

test.js

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

// to parse request body
var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({extended: false}));
app.use(bodyParser.json());
```

The request's query data is retrieved as shown blow. The data is shown back to the user.

test.js

```
// Request query data
app.get('/data', function(req, res) {
  var queryData  = JSON.stringify(req.query);
  res.send("Query Data " +  queryData);
});
```

The output for a sample request with two parameters in the request query is shown below.

The request's path data as well as the query data is retrieved as shown blow. The data from the two sources is shown back to the user.

test.js

```
// Request params data and query data
app.get('/data/:id1/:id2', function(req, res) {
  var paramsData = JSON.stringify(req.params);
  var queryData  = JSON.stringify(req.query);
  res.send("Params Data " + paramsData + "<br/>" +
           " Query Data " +  queryData);
});
```

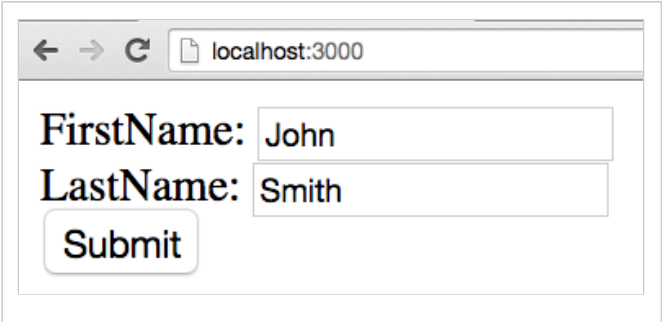
The output for a sample request with two path parameters and two parameters in the request query is shown below.

Similarly, the form data can be processed to access the body parameters as shown below. The default GET request shows the form to the user. When the form is submitted using the POST method, the input data names and values are retrieved from the request's *body* property.

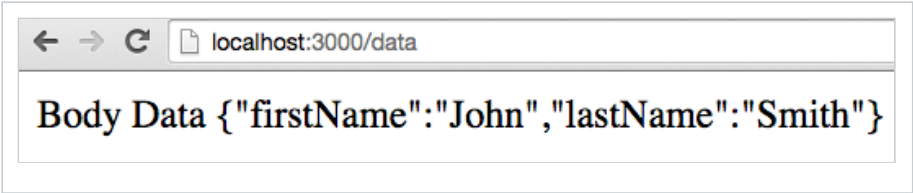
test.js

```
22
23  app.get('/', function(req, res){
24    var html =
25     '<form method="POST" action="/data">' +
26       'FirstName: <input name="firstName" value=""><br/>' +
27       'LastName: <input name="lastName" value=""><br/>' +
28       '<input type="Submit"></form>'
29    res.send(html);
30  });
31
32  // request body data
33  app.post('/data', function(req, res) {
34    var bodyData  = JSON.stringify(req.body);
35    res.send("Body Data " +  bodyData);
36  })
```

The initial form is shown to the user as shown below.



After the user submits the data, the application shows the response back with the provided values.



Case Study – Course Manager with REST API

The following case study builds a REST API for consuming data in the JSON format from the Express web application. The following module exports the functions that manage the data for the case study. Since the module can be used standalone, the functions return a copy of the data rather than references to the data objects.

The getCourse() function returns a new object with the data corresponding to the specified course *id*. The getCourses() function returns a new array with deep clone of the *courses*.image 2

ex08_courses.js

```
var courses = [
 {id: 'cs601', name: 'Web Application Development'},
 {id: 'cs602', name: 'Server-Side Web Development'},
 {id: 'cs701', name: 'Rich Internet Appl Development'}
];

exports.getCourse = function(course_id) {
  for (var i = 0; i < courses.length; i++) {
    if (courses[i].id == course_id) {
      return {id: courses[i].id, name : courses[i].name};
    }
  }
  return {id: course_id, name : 'Unknown'};
};


// return a deep clone
exports.getAllCourses = function() {
  var result = JSON.parse(JSON.stringify(courses));
  return result;
}

```

The function addCourse() adds a new course object to the *courses* array, if the specified course *id* is not present. Otherwise, the operation is ignored. The function removeCourse() removes the corresponding course object from the *courses* array, if the specified course *id* is present. Otherwise, the operation is ignored.

ex08_courses.js

```

exports.addCourse = function(course_id, course_name) {
  for (var i = 0; i < courses.length; i++) {
    if (courses[i].id == course_id) {
      break;
    }
  }
  if (i == courses.length) {
    courses.push({id: course_id, name : course_name});
  };
}

exports.removeCourse = function(course_id) {
  for (var i = 0; i < courses.length; i++) {
    if (courses[i].id == course_id) {
      break;
    }
  };
  if (i != courses.length) {
    courses.splice(i, 1);
  };
}
```

The Express application using Handlebars for the view engine is setup as shown below.

ex08_rest.js

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

// to parse request body
var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

// static resources
app.use(express.static(__dirname + '/public'));

// setup handlebars view engine
var handlebars = require('express-handlebars');
app.engine('handlebars', handlebars());
app.set('view engine', 'handlebars');

```

The routes are configured for the paths as illustrated in the following table:

Route	HTTP	**Function**
/api/courses	GET	Return all the courses as array of JSON objects
/api/course/:cid	GET	Return the specified course as a JSON object
/api/course	POST	Add the new course to the courses data
/api/course/:cid	DELETE	Delete the course with the specified id from the courses data

The Express application with the above routes is setup as shown below. The handlers return the data is the JSON format. When a course is deleted, or when a course is added, the modified data is returned as the response.

ex08_rest.js

```
17  // module
18  var courses = require('./ex08_courses');
19
20  app.get('/api/courses', function(req, res){
21    res.json(courses.getAllCourses())
22  });
23
24  app.get('/api/course/:cid', function(req, res){
25    res.json(courses.getCourse(req.params.cid))
26  });
27
28  app.delete('/api/course/:cid', function(req, res){
29    courses.removeCourse(req.params.cid);
30    res.json(courses.getAllCourses());
31  });
32
33  app.post('/api/course', function(req, res) {
34    courses.addCourse(req.body.cid, req.body.cname);
35    res.json(courses.getAllCourses());
36  });
```

```
37
38  app.get('/new', function (req, res){
39    res.render('newCourse');
40  });
```

The following form is used for adding the new course information via the HTTP POST method.

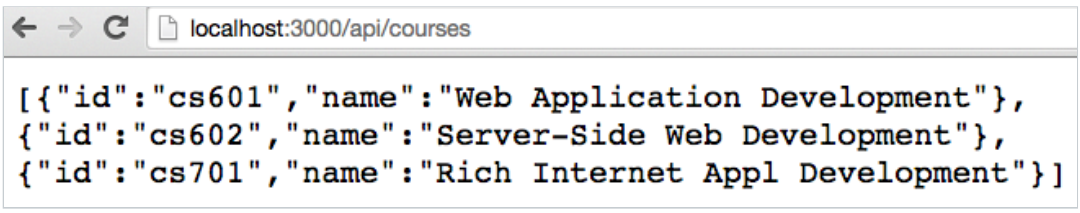
newCourse.handlebars

```
<form method="POST" action="/api/course">

  <table>
  <tr>
   <td valign="top">
    <label for="cid">Course Id:</label>
   </td>
   <td valign="top">
    <input  type="text" name="cid" size="30">
   </td>
  </tr>
  <tr>
   <td valign="top">
    <label for="cname">Course Name:</label>
   </td>
   <td valign="top">
    <input  type="text" name="cname" size="30">
   </td>
  </tr>
  <tr>
   <td colspan="2">
    <input type="submit" width="100"/>
   </td>
  </tr>
 </table>

</form>
```

The *curl* utility can be used for testing the HTTP operations. The GET request for retrieving the list of all courses is shown below.

```
>curl -X GET "http://localhost:3000/api/courses";echo
[{"id":"cs601","name":"Web Application Development"},
{"id":"cs602","name":"Server-Side Web Development"},{
"id":"cs701","name":"Rich Internet Appl Development"}
]
```

The same data can also be retrieved through the browser using the GET request shown below.



Information about a specific course is retrieved using *curl* as shown below.

```
>curl -X GET "http://localhost:3000/api/course/cs602";echo
{"id":"cs602","name":"Server-Side Web Development"}
```

The same data can also be retrieved through the browser using the GET request shown below.

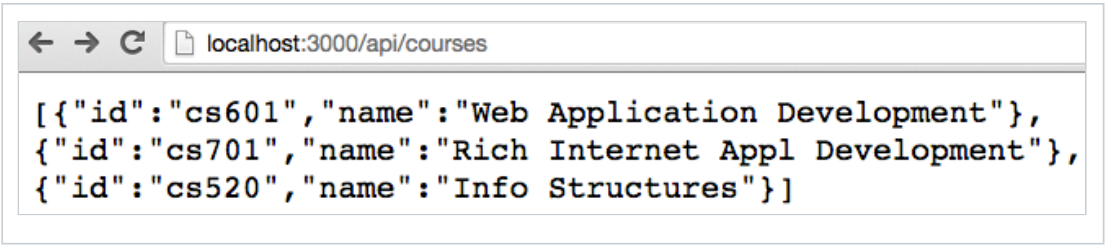
For deleting a course, the HTTP DELETE request is sent through *curl* as shown below. The returned data is the remaining list of courses.

```
>curl -X DELETE "http://localhost:3000/api/course/cs602";echo
[{"id":"cs601","name":"Web Application Development"},{"id":"cs
701","name":"Rich Internet Appl Development"}]
```

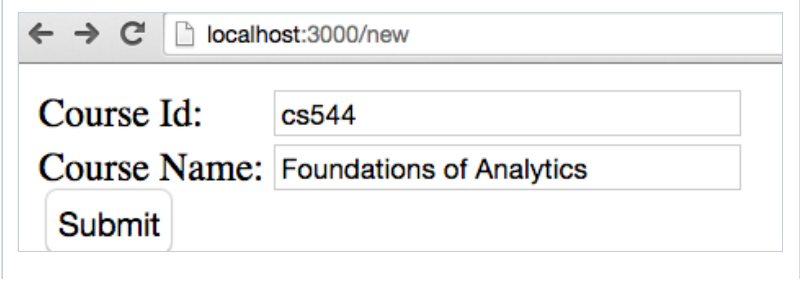
New data can be sent through the POST request using *curl*. The data for the new course is sent as JSON data. The Express *bodyParser* is configured to parse JSON body data as well. The new list of courses is returned as the response.

```
>curl -X POST -H "Content-type: application/json" \
> "http://localhost:3000/api/course" \
> -d '{"cid":"cs520", "cname":"Info Structures"}';echo
[{"id":"cs601","name":"Web Application Development"},{
"id":"cs701","name":"Rich Internet Appl Development"},
{"id":"cs520","name":"Info Structures"}]
```

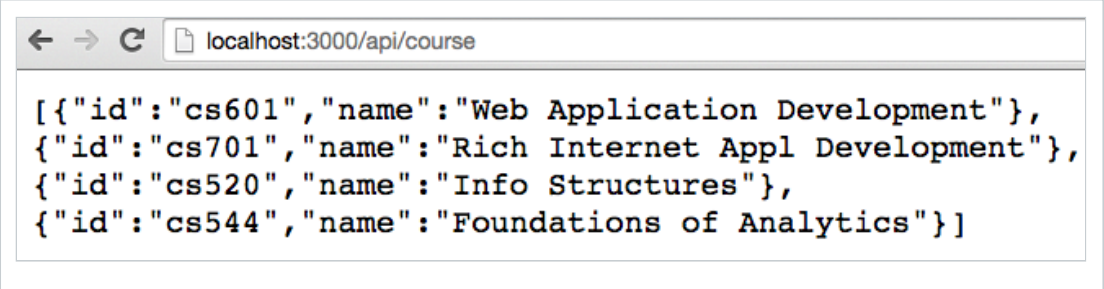
After the above insertion of the specified course, the current data can also be viewed through the browser using the GET request shown below.



The data can also be added using the HTML form configured in the Express application.



After the form is submitted, the new list of courses is returned as a response as shown below.



After the above insertion of the specified course, the data for the newly added course can also be viewed through the browser using the GET request shown below.

Multiple MIME formats

The REST APIs typically provide the option to the users for sending the requested data in different MIME formats. Typical formats used are JSON, XML, HTML, or plain text. The following Express application configures the same route to handle multiple MIME type requests. The format() method of the *response* object is used for specifying the various MIME types and how they have to handled. In the following example, the list of courses is sent either in JSON format, XML format, HTML format, or plain text format.

ex09_rest.js

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

var courses = require('./ex08_courses');

app.get('/api/courses', function(req, res){

  res.format({

    'application/json': function() {
      res.json(courses.getAllCourses());
    },

    'application/xml': function() {
      var coursesXml =
        '<?xml version="1.0"?>\n<courses>\n' +
          courses.getAllCourses().map(function(c){
            return ' <course id="' + c.id + '">' +
              c.name + '</course>';
          }).join('\n') + '\n</courses>\n';

      res.type('application/xml');
      res.send(coursesXml);
    },
```

ex09_rest.js

```
25
26      'text/html': function() {
27        var coursesHtml = '<ul>\n' +
28          courses.getAllCourses().map(function(c){
29            return ' <li>' + c.id + ' - ' +
30                        c.name + '</li>';
31          }).join('\n') + '\n</ul>\n';
32
33        res.type('text/html');
34        res.send(coursesHtml);
35      },
36
37      'text/plain': function() {
38        var coursesText =
39          courses.getAllCourses().map(function(c){
40            return c.id + ': ' + c.name;
41          }).join('\n') + '\n';
42
43        res.type('text/plain');
44        res.send(coursesText);
45      },
46
47      'default': function() {
48        res.status(404);
49        res.send("<b>404 - Not Found</b>");
50      }
51    });
52  });
```

The Accept HTTP header coming as part of the request is used to determine which MIME type is requested by the user. With *curl*, the default value is *application/json*.

```
>curl -X GET "http://localhost:3000/api/courses";echo
[{"id":"cs601","name":"Web Application Development"},{
"id":"cs602","name":"Server-Side Web Development"},{"i
d":"cs701","name":"Rich Internet Appl Development"}]
```

The XML formatted response is requested explicitly through *curl* by sending the header as shown below.

```
>curl -X GET -H "Accept:application/xml"  \
> "http://localhost:3000/api/courses"
<?xml version="1.0"?>
<courses>
 <course id="cs601">Web Application Development</course>
 <course id="cs602">Server-Side Web Development</course>
 <course id="cs701">Rich Internet Appl Development</course>
</courses>
```

The HTML formatted response is requested explicitly through *curl* by sending the header as shown below.

```
>curl -X GET -H "Accept:text/html"  \
>   "http://localhost:3000/api/courses"
<ul>
 <li>cs601 - Web Application Development</li>
 <li>cs602 - Server-Side Web Development</li>
 <li>cs701 - Rich Internet Appl Development</li>
</ul>
```

Similarly, the plain text formatted response is requested explicitly through *curl* by sending the header as shown below.

```
>curl -X GET -H "Accept:text/html"  \
>   "http://localhost:3000/api/courses"
<ul>
 <li>cs601 - Web Application Development</li>
 <li>cs602 - Server-Side Web Development</li>
 <li>cs701 - Rich Internet Appl Development</li>
</ul>
```

If the request is issued from a web browser, the default request header is *application/html* as shown in the case below.

For Express applications that require HTTP cookie processing, the *cookie-parser* module is installed for the application as shown below.

```
>npm install --save cookie-parser
```

Similarly, if HTTP session handling is required, the *cookie-parser* and *express-session* modules are required for the application.

```
>npm install --save express-session
```

The following example demonstrates the usage of cookies in a web application. Node allows both unsigned cookies as well as signed cookies that get saved by the browser. A secret key is required for signing the cookies. For the example, the following module exports the *cookieSecret* property.

credentials.js

```
module.exports = {
   cookieSecret: 'supersecret'
}
```

The Express application for handling cookies is configured as shown below. The cookie-parser module is loaded and initialized using the secret key.

ex01_cookies.js

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

// setup handlebars view engine
var handlebars = require('express-handlebars');
app.engine('handlebars', handlebars());
app.set('view engine', 'handlebars');

// for signed cookies
var credentials = require('./credentials.js');

var cookieParser = require('cookie-parser');
app.use(cookieParser(credentials.cookieSecret));

```

The web application provides the following route for adding a cookie as part of the response to the browser. The name and value for the cookie are obtained from the request query object.

ex01_cookies.js

```
// unsigned cookies
app.get('/add', function(req, res) {
   var name = req.query.name;
   var value = req.query.value;
   if (name) {
      res.cookie(name, value);
   }
   res.redirect("/");
});
```

For adding a signed cookie, the signed option is set to *true*. By default, all cookies expire when the browser terminates the session. The expiration date for the cookies (signed and unsigned) can also be set using the maxAge option.

ex01_cookies.js

```
24
25  // signed cookies
26  app.get('/secure', function(req, res) {
27    var name = req.query.name;
28    var value = req.query.value;
29    if (name) {
30      res.cookie(name, value,
31              {signed: true,
32                maxAge: 24*60*60*1000});
33    }
34    res.redirect("/");
35  });
```

After the cookies are added, the response redirects the browser to the default GET request. The route is handled by retrieving all the unsigned and signed cookies from the request object and passed the context to the view page as shown below.

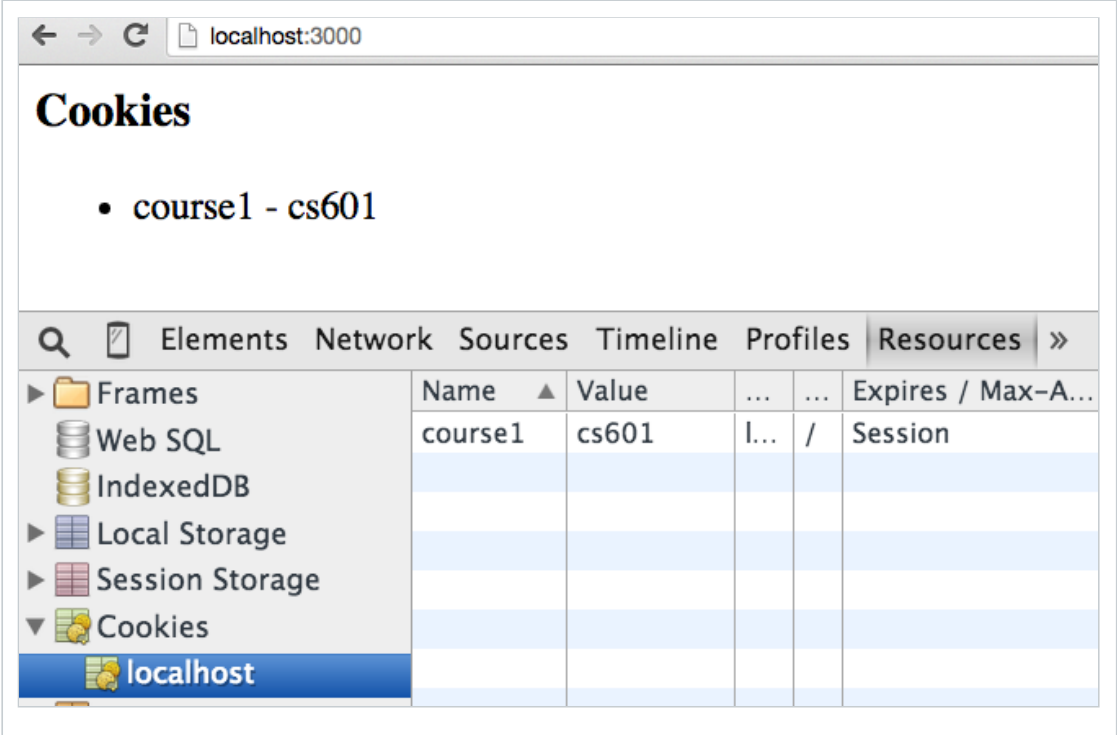
ex01_cookies.js

```
36
37  // show all cookies
38  app.get('/', function(req, res) {
39    var result = [];
40    for (var key in req.cookies) {
41      result.push({name: key,
42          value: req.cookies[key]});
43    };
44    for (var key in req.signedCookies) {
45      result.push({name: key,
46          value: req.signedCookies[key]});
47    };
48    res.render('showCookies', {cookies: result});
49  });
```

A cookie can be added using the link shown below.

http://localhost:3000/add?name=course1&value=cs601

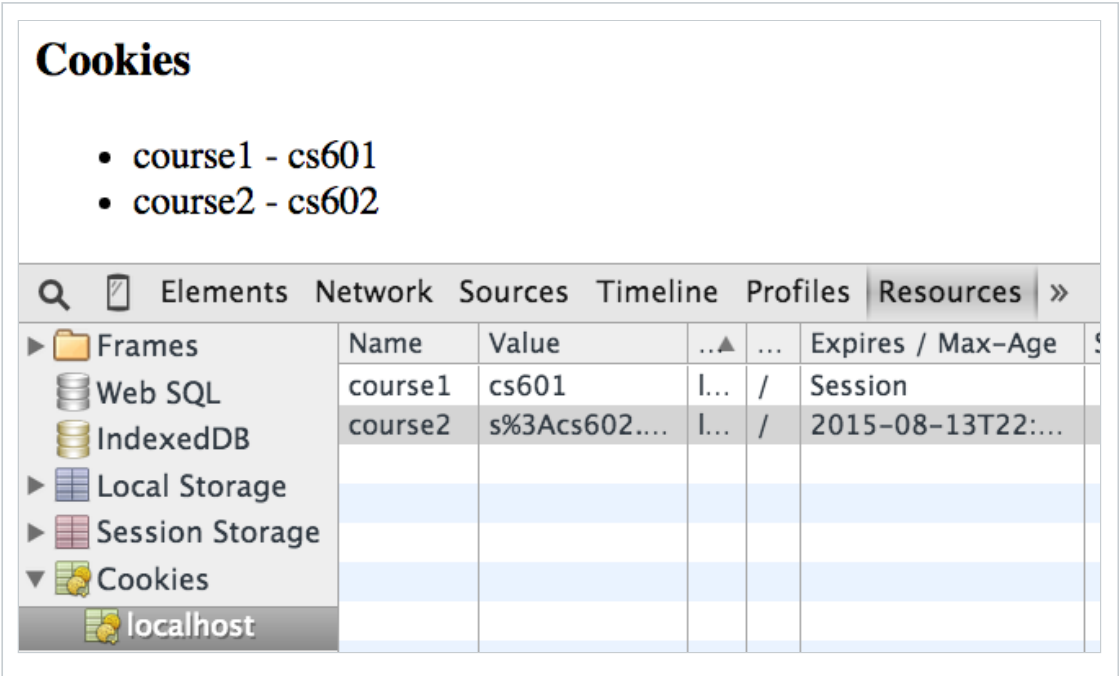
The response redirects the user to the view page showing all the existing cookies. The cookie information can also be viewed in the Chrome developer tools as shown below.



The following request adds a signed cookie.

http://localhost:3000/secure?name=course2&value=cs602

The resulting page shows all the existing cookies. The browser only stores the signed version of the cookie. When the value is retrieved in the web application, the value is automatically decrypted.



For enabling HTTP session handling, the *cookie-parser* module and the *express-session* modules are loaded and configured as shown below. The session *id* is stored as a cookie by the browser. Hence, that *id* is signed. The rest of application's data is now associated with the session object.

ex02_session.js

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

// setup handlebars view engine
var handlebars = require('express-handlebars');
app.engine('handlebars', handlebars());
app.set('view engine', 'handlebars');

// to parse request body
var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

// for signed cookies
var credentials = require('./credentials.js');
var cookieParser = require('cookie-parser');
var expressSession = require('express-session');

// cookie-parser first
app.use(cookieParser());
app.use(expressSession(
  {secret: credentials.cookieSecret}));

```

The following application keeps track of a list of tasks that the user wishes to do in the session object. The taskList property of the session object is set to an empty array initially, and the accumulated data is added to this array when a new task is submitted.

ex02_session.js

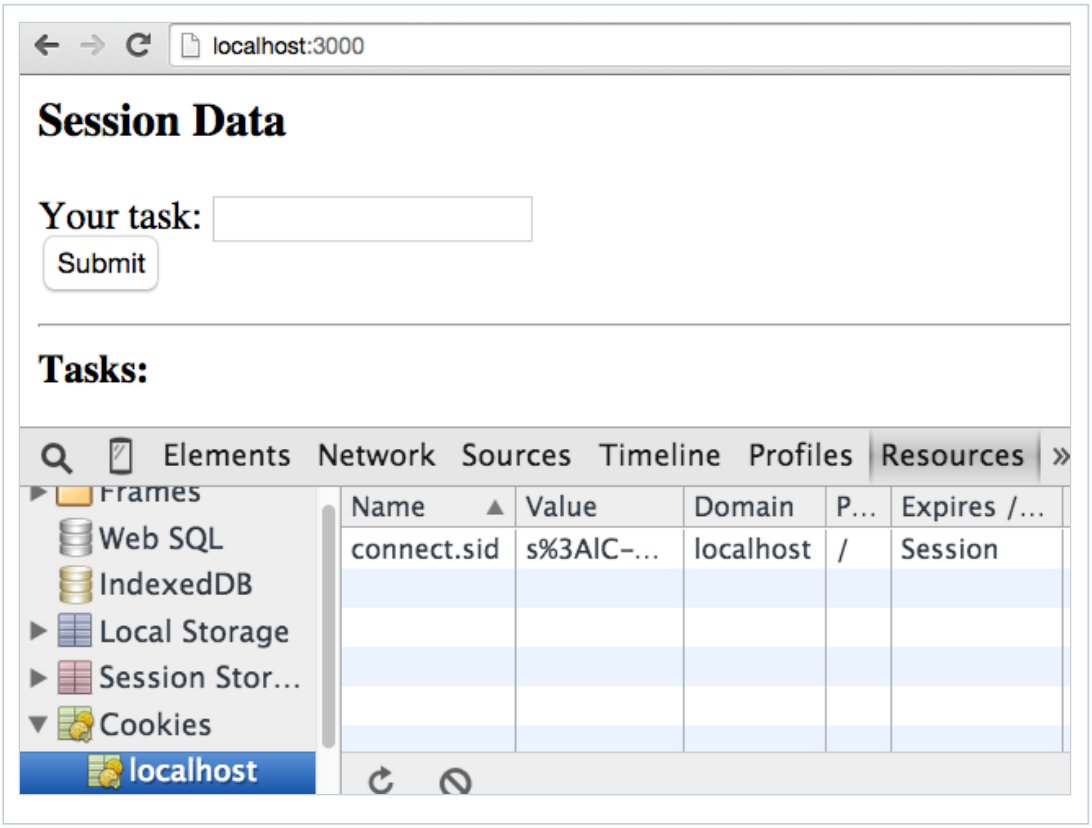
```
// show session data
app.get('/', function(req, res) {
  if (req.session.taskList === undefined) {
    req.session.taskList = [];
  }
  res.render('showSession',
    {data: req.session.taskList});
});

// handle form submission
app.post('/', function(req, res){
  req.session.taskList.push(req.body.task);
  res.redirect('/');
});

```

The following view provides the option to the user to submit a new task and also shows the current tasks stored in the session.

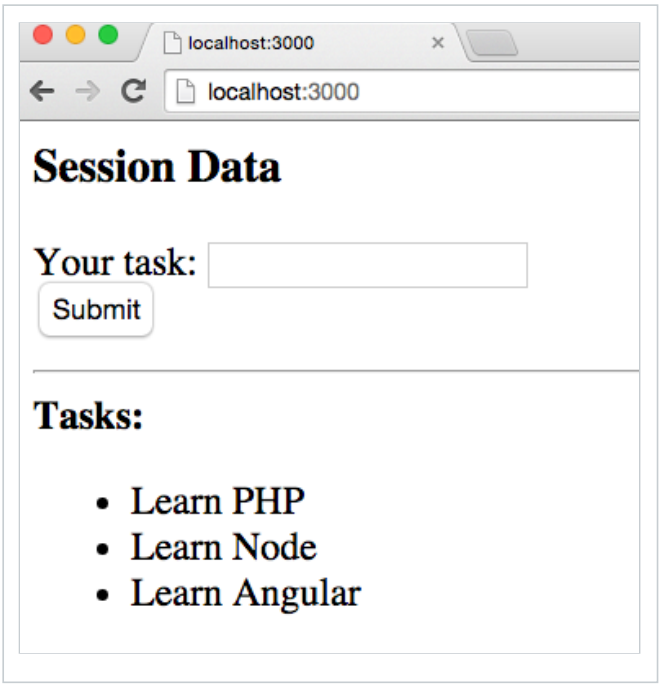
showSession.handlebars

```
<h3>Session Data</h3>
<form action="/" method="post">
  Your task: <input type="text" name="task"><br>
  <button type="submit">Submit</button>
 </form>
 <hr>
<b>Tasks:</b>
<ul>
    {{#each data}}
      <li>{{this}}</li>
    {{/each}}
</ul>
```

The application is accessed as shown below. The session data uses the cookie connect.sid to associate the session for the user.



After tasks are submitted, the web application shows the accumulated data as shown below.



Bibliography

Express: Fast, unopinionated, minimalist web framework for Node.js, http://expressjs.com

Handlebars, http://handlebarsjs.com

Node body-parser module, https://github.com/expressjs/body-parser

Express cookie-parser module, https://github.com/expressjs/cookie-parser

Express session module, https://github.com/expressjs/session

Ethan Brown, Web Development with Node and Express, O'Reilly, 2014.