Module 3

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 3 Study Guide and Deliverables

Readings:	Brown: Chapters 13-17
Discussions:	Discussion 3 Due: Tuesday, July 24 at 6:00 AM ET
Assignments:	Assignment 3 Due: Tuesday, July 24 at 6:00 AM ET
Live Classrooms:	Wednesday, July 18 from 8:00-10:00 PM ET

Persistence using MySQL

Objectives

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

- Write standalone MySQL database applications
- Develop Node web applications that persist and interact with data form MySQL database

Introduction

MySQL is one of the popular relational open source databases. The instructions for installing MySQL can be found on the web site:

http://dev.mysql.com/downloads/mysql/

The community edition of MySQL is freely available and is used in this lecture. A default database named, *test*, is provided by the installation. The examples use the credentials *root* and the empty string for the username and password, respectively.

For the *Node* applications to interact with the MySQL database, the *mysql* module is required and is installed for the applications as shown below. This is the *Node.js* driver for MySQL written in JavaScript.

```
>npm install --save mysql
```

Connecting to MySQL

A connection is required for accessing the database. The connection object is created using the createConnection() method from the *mysql* module. The method takes an object as its argument with the properties shown below.

ex01_mysql.js

```
var mysql = require("mysql");
var credentials = require("./credentials.js");

var connection = mysql.createConnection({
  "host": "localhost", "port": 3306,
  "user": credentials.username,
  "password": credentials.password,
  "database": "test"
});
```

Using the connection object, the connection to the database is established using the connect() method. The argument for this method is the callback function that is invoked after the connection is established. After the database operations are completed, the connection can be closed using the end() method. This method closes the connection gracefully by letting any queued queries to execute completely. An alternate approach is to use the destroy() method. This method immediately closes the database connection. The following example illustrates the typical usage of the connect() and end() methods.

ex01_mysql.js

```

connection.connect(function(error) {
  if (error) {
    console.error(error);
    return;
  }
  // Connection successfully established
  console.log("Connected...");
});

// Do database operations

connection.end(function(error) {
  if (error) {
    console.error(error);
    return;
  }
  // Connection successfully closed
  console.log("Closed...");
});
```

Executing the above program results in the following output in the successful scenario.

```
>node ex01_mysql.js
Connected...
Closed...
```

If the MySQL instance is not running, the above program results in an error while trying to establish a connection, as shown below.

```
>node ex01_mysql.js
{ [Error: connect ECONNREFUSED]
  code: 'ECONNREFUSED',
  errno: 'ECONNREFUSED',
  syscall: 'connect',
  fatal: true }
```

If incorrect credentials are used when MySQL instance is running, the above program results in an error while trying to establish a connection, as shown below.

```
>node ex01_mysql.js
{ [Error: ER_ACCESS_DENIED_ERROR: Access denied for user
'root'@'localhost' (using password: YES)]
  code: 'ER_ACCESS_DENIED_ERROR',
  errno: 1045,
  sqlState: '28000',
  fatal: true }
```

Executing Database Queries – SELECT Statement

The following schema and initial data shown is used for demonstrating the execution of SQL queries.

createdb.sql

```
1  drop table if exists Weather;
2  create table Weather (
3    city varchar(100) not null,
4    temperature integer not null
5  );
6
7  insert into Weather (city, temperature) values ('Austin', 48);
8  insert into Weather (city, temperature) values ('Baton Rouge', 57);
9  insert into Weather (city, temperature) values ('Jackson', 50);
10 insert into Weather (city, temperature) values ('Montgomery', 53);
11 insert into Weather (city, temperature) values ('Phoenix', 67);
12 insert into Weather (city, temperature) values ('Sacramento', 66);
13 insert into Weather (city, temperature) values ('Santa Fe', 27);
14 insert into Weather (city, temperature) values ('Tallahassee', 59);
```

For executing SQL queries, after connecting to the database, the connection object's query() method is used. The two arguments for this method are the SQL string to execute and the callback function. After the query is executed, the callback function is invoked with the possible results or an error.

The following example shows the selection of all the rows from the *Weather* table. On successful completion, the results are available as an array of objects. Each object contains the information for each matched row of the query. The selected columns from the table are mapped as properties of these objects.

ex02_mysql.js

```
19
20  // Do database operations
21
22  connection.query("Select * from Weather",
23    function(error, rows) {
24      if (error) {
25        console.error(error);
26        return;
27      }
28      console.log(rows);
29    });
```

Executing the above program results in the following output in the successful scenario.

```
>node ex02_mysql.js
Connected...
[ { city: 'Austin', temperature: 48 },
  { city: 'Baton Rouge', temperature: 57 },
  { city: 'Jackson', temperature: 50 },
  { city: 'Montgomery', temperature: 53 },
  { city: 'Phoenix', temperature: 67 },
  { city: 'Sacramento', temperature: 66 },
  { city: 'Santa Fe', temperature: 27 },
  { city: 'Tallahassee', temperature: 59 } ]
Closed...
```

The following example iterates over the results, accesses the *city* and *temperature* properties for each result, and prints a custom output to the console.

ex03_mysql.js

```
20  // Do database operations
21
22  connection.query("Select * from Weather",
23    function(error, rows) {
24      if (error) {
25        console.error(error);
26        return;
27      }
28      for (var i = 0; i < rows.length; i++) {
29        console.log(rows[i].city, "-",
30                    rows[i].temperature + " degrees");
31      }
32    });
```

The output from the above program is shown below.

```
>node ex03_mysql.js
Connected...
Austin - 48 degrees
Baton Rouge - 57 degrees
Jackson - 50 degrees
Montgomery - 53 degrees
Phoenix - 67 degrees
Sacramento - 66 degrees
Santa Fe - 27 degrees
Tallahassee - 59 degrees
Closed...
```

Executing Database Queries – INSERT Statement

For inserting new data into the table in the database, the INSERT prepared statement is used. If the data is available as object, the placeholder in the query is used to set the specified data as shown in the following example.

ex04_mysql.js

```
20  // Do database operations
21
22  var newData = {city: 'Oklahoma City', temperature: 50};
23
24  connection.query("Insert into Weather SET ?", newData,
25    function(error, result) {
26      if (error) {
27        console.error(error);
28        return;
29      }
30      console.log(result);
31    });
```

If no error occurs, the callback function's result is an object with the structure as shown below. The *affectedRows* property shows the number of rows affected in the database as a result of this operation.

```
>node ex04_mysql.js
Connected...
{ fieldCount: 0,
  affectedRows: 1,
  insertId: 0,
  serverStatus: 2,
  warningCount: 0,
  message: '',
  protocol41: true,
  changedRows: 0 }
Closed...
```

Executing the program selecting all the rows shows the new data that got inserted into the table as illustrated below.

```
>node ex03_mysql.js
Connected...
Austin - 48 degrees
Baton Rouge - 57 degrees
Jackson - 50 degrees
Montgomery - 53 degrees
Phoenix - 67 degrees
Sacramento - 66 degrees
Santa Fe - 27 degrees
Tallahassee - 59 degrees
Oklahoma City - 50 degrees
Closed...
```

Executing Database Queries – UPDATE Statement

For updating the current data in the database, the UPDATEprepared statement is used with placeholders. If the data is available as an array, the placeholders in the query are used to set the specified data as shown in the following example. The order of the data in the array must match the order of the placeholders in the query.

ex05_mysql.js

```
20  // Do database operations
21
22  var newData = [60, 'Oklahoma City'];
23
24  connection.query(
25    "Update Weather SET temperature = ? Where city = ?",
26    newData,
27    function(error, result) {
28      if (error) {
29        console.error(error);
30        return;
31      }
32      console.log(result);
33    });
```

If no error occurs, the callback function's result is an object with the structure as shown below. The *affectedRows* and *changedRows* properties show the number of rows affected in the database as a result of this operation.

```
>node ex05_mysql.js
Connected...
{ fieldCount: 0,
  affectedRows: 1,
  insertId: 0,
  serverStatus: 34,
  warningCount: 0,
  message: '(Rows matched: 1  Changed: 1  Warnings: 0',
  protocol41: true,
  changedRows: 1 }
Closed...
```

Executing the program selecting all the rows shows the updated from the table as illustrated below.

```
>node ex03_mysql.js
Connected...
Austin - 48 degrees
Baton Rouge - 57 degrees
Jackson - 50 degrees
Montgomery - 53 degrees
Phoenix - 67 degrees
Sacramento - 66 degrees
Santa Fe - 27 degrees
Tallahassee - 59 degrees
Oklahoma City - 60 degrees
Closed...
```

Executing Database Queries – DELETE Statement

For deleting specific data in the database, the DELETEprepared statement is used with placeholders, if necessary. If the data is available as an array, the placeholders in the query are used to set the specified data as shown in the following example. The order of the data in the array must match the order of the placeholders in the query, if more than one field is needed in the query.

ex06_mysql.js

```
20  // Do database operations
21
22  var newData = ['Oklahoma City'];
23
24  connection.query(
25    "Delete from Weather Where city = ?",
26    newData,
27    function(error, result) {
28      if (error) {
29        console.error(error);
30        return;
31      }
32      console.log(result);
33    });
```

If no error occurs, the callback function's result is an object with the structure as shown below. The *affectedRows* property shows the number of rows affected in the database as a result of this operation.

```
>node ex06_mysql.js
Connected...
{ fieldCount: 0,
  affectedRows: 1,
  insertId: 0,
  serverStatus: 34,
  warningCount: 0,
  message: '',
  protocol41: true,
  changedRows: 0 }
Closed...
```

Executing the program selecting all the rows shows the updated from the table as illustrated below.

```
>node ex03_mysql.js
Connected...
Austin - 48 degrees
Baton Rouge - 57 degrees
Jackson - 50 degrees
Montgomery - 53 degrees
Phoenix - 67 degrees
Sacramento - 66 degrees
Santa Fe - 27 degrees
Tallahassee - 59 degrees
Closed...
```

Transaction support is available through the connection object's beginTransaction(), commit(), and rollback() methods.

Connection Pooling

If frequent connections are required to the database, creating a database connection pool is more efficient. The connection pool object is created using the createPool() method from the *mysql* module. The method takes an object as its argument with the properties shown below.

ex07_mysql.js

```
1  var mysql = require("mysql");
2  var credentials = require("./credentials.js");
3
4  var pool = mysql.createPool({
5    "host": "localhost", "port": 3306,
6    "user": credentials.username,
7    "password": credentials.password,
8    "database": "test"
9  });
10
```

Each time a new connection is required, the connection is obtained from the pool. Once the operations are done, the connection can be returned to the pool. The createPool() method also supports additional options for the connectionLimit and queueLimit. The connectionLimit property defaults to 10 and specifies the maximum number of connections that can be created at once. The queueLimit property specifies the maximum number of connection requests that can be queued by the pool. The default value is 0, indicating no limit.

The connection pool's getConnection() method is used for requesting a database connection. The callback function specified as its argument provides the requested connection object, as shown below. The release() method on the connection object returns the connection to the pool.

ex07_mysql.js

```
11  pool.getConnection(function(error, connection) {
12    if (error) {
13      console.error(error);
14      return;
15    }
16    // Connection successfully established
17    console.log("Connection from pool...");
18
19    // Do database operations
20
21    connection.query("Select * from Weather",
22    function(error, rows) {
23      if (error) {
24        console.error(error);
25        return;
26      }
27      console.log(rows);
28    });
29
30    // Release the connection to the pool
31    connection.release();
32
33  });
```

Executing the program selecting all the rows shows the data from the table as illustrated below.

```
>node ex07_mysql.js
Connection from pool...
[ { city: 'Austin', temperature: 48 },
  { city: 'Baton Rouge', temperature: 57 },
  { city: 'Jackson', temperature: 50 },
  { city: 'Montgomery', temperature: 53 },
  { city: 'Phoenix', temperature: 67 },
  { city: 'Sacramento', temperature: 66 },
  { city: 'Santa Fe', temperature: 27 },
  { city: 'Tallahassee', temperature: 59 } ]
```

Escaping User Input

In order to avoid the SQL Injection attacks, care should be taken while handling any input data provided by the user. Any user provided data should be escaped using the mysql.escape(), connection.escape(), or pool.escape() methods as shown below.

```
var input = 'Austin';
connection.query("Select * from Weather WHERE city = " +
                 connection.escape(input),
  function(error, rows) {
    if (error) {
      console.error(error);
      return;
    }
    console.log(rows);
  });
```

The alternative is to use ? characters as placeholders in the query as shown below.

```
var input = 'Austin';
connection.query("Select * from Weather WHERE city = ?",
                 [input],
  function(error, rows) {
    if (error) {
      console.error(error);
      return;
    }
    console.log(rows);
  });
```

Case Study – Course Manager

The following example shows the web application to manage the list of courses. The schema and the initial data for the database are shown below.

courses.sql

```
drop table if exists met_courses;

create table met_courses (
  id int(11) NOT NULL AUTO_INCREMENT,
  course_number varchar(32) NOT NULL,
  course_name   varchar(200) NOT NULL,
  PRIMARY KEY (id)
);

INSERT INTO met_courses (id, course_number, course_name) VALUES
    (1, 'cs520', 'Information Structures'),
    (2, 'cs601', 'Web Application Development'),
    (3, 'cs602 ', 'Server-Side Web Development');
```

The web application provides the capability for displaying all the courses, edit or delete an existing course, and adding a new course. The web application uses Express and Express Handlebars as the view template engine. The Express Router is used for specifying the end points and their handlers, as shown below.

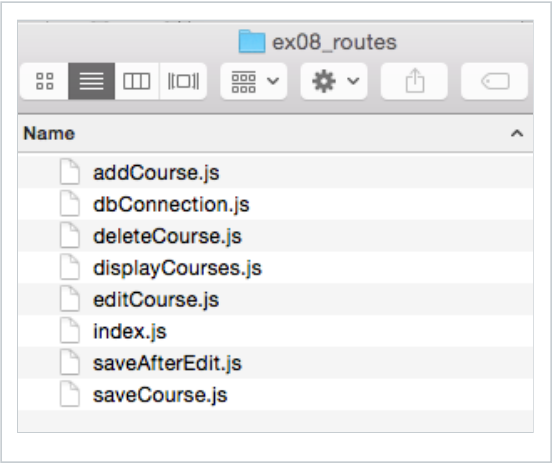
ex08_caseStudy.js

```
1  var express = require('express');
2  var bodyParser = require('body-parser');
3  var handlebars = require('express-handlebars');
4
5  var app = express();
6
7  // setup handlebars view engine
8  app.engine('handlebars',
9      handlebars({defaultLayout: 'main_logo'}));
10 app.set('view engine', 'handlebars');
11
12 // static resources
13 app.use(express.static(__dirname + '/public'));
14
15 app.use(bodyParser.json());
16 app.use(bodyParser.urlencoded({ extended: false }));
17
18 // Routing
19 var routes = require('./ex08_routes/index');
20 app.use('/', routes);
21
```

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

Route	HTTP	**Function**
/courses	GET	Display all the courses from the database
/courses/add	GET	Show form to add a new course
/courses/add	POST	Save the new course to the database
/courses/edit/:id	GET	Show form with the data about the course with the specified id
/courses/edit/:id	POST	Update the course data in the database for the specified id
/courses/delete/:id	GET	Delete the course with the specified id from the database

Each of the above operations is specified in their own modules as shown in the following structure.



The *index.js* file creates the router with the routes and their handlers as shown below. The module exported by this file is used in the main web application.

index.js

```
var express = require('express');
var router = express.Router();

// other modules
var displayCourses  = require("./displayCourses");
var addCourse       = require("./addCourse");
var saveCourse      = require("./saveCourse");
var editCourse      = require("./editCourse");
var saveAfterEdit   = require("./saveAfterEdit");
var deleteCourse    = require("./deleteCourse");

// router specs
router.get('/', function(req, res, next) {
  res.redirect('/courses');
});

router.get('/courses',             displayCourses);

router.get('/courses/add',         addCourse);
router.post('/courses/add',        saveCourse);

router.get('/courses/edit/:id',    editCourse);
router.post('/courses/edit/:id',   saveAfterEdit);

router.get('/courses/delete/:id', deleteCourse);

module.exports = router;
```

The *displayCourses* module exports the function that retrieves all the data from the database table passes the data to the view to be rendered as shown below.

displayCourses.js

```
var connection =
  require('./dbConnection.js').dbConnect();

module.exports =
  function displayCourses(req , res , next){
    connection.query('select * from met_courses',
      function(err , rows){
        if(err)
            console.log("Error Selecting : %s ",err);
        res.render('displayCoursesView',
          {title:"List of Courses", data:rows});
    });
};
```

The view page for displaying all the course data is shown below. In addition to the course name and course number, links are provided for editing and deleting each course. The view also has the button for adding a new course.

displayCoursesView.handlebars

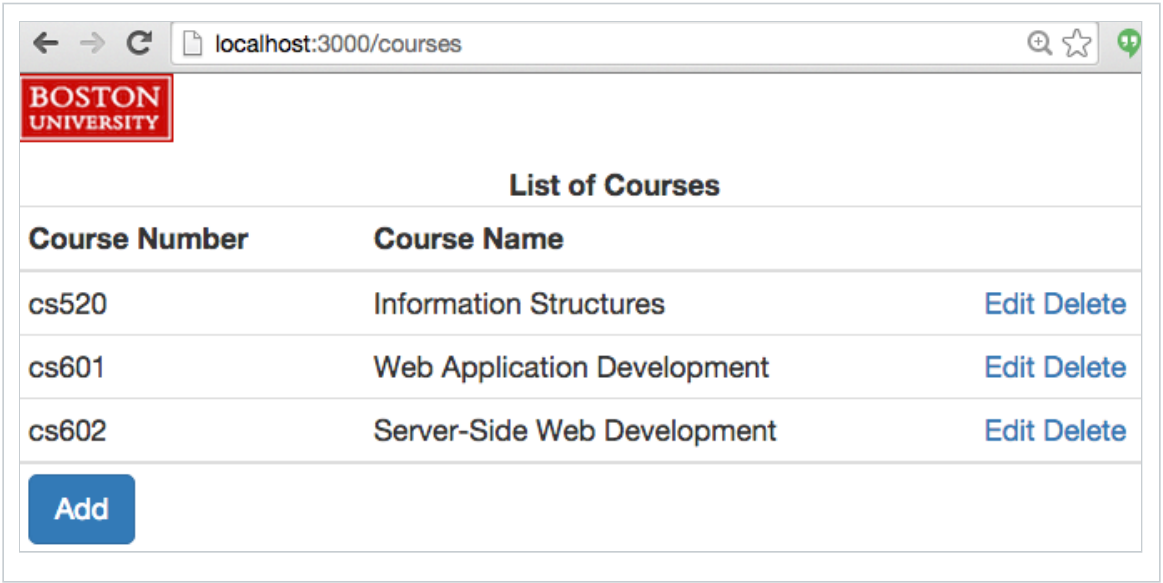
```
<div>
  <div class="table-responsive">
    <table class="table table-condensed">
      <thead>
        <tr>
          <th>Course Number</th>
          <th>Course Name</th>
          <th></th>
        </tr>
      </thead>
      <tbody>
        {{#each data}}
          <tr>
            <td>{{this.course_number}}</td>
            <td>{{this.course_name}}</td>
            <td>
              <a href="/courses/edit/{{this.id}}">Edit</a>
              <a href="/courses/delete/{{this.id}}">Delete</a>
            </td>
          </tr>
        {{/each}}
      </tbody>
    </table>
    <button class="btn btn-primary" onClick="addCourse();">Add</button>
  </div>
</div>
```

The following layout is used for all the views in the application.

main_logo.handlebars

```
<!doctype html>
<html>
<head>
  <title>CS602</title>
  <link rel='stylesheet' href='/stylesheets/bootstrap.css' />
  <script src="/javascripts/clickActions.js"
    type="text/javascript"></script>
</head>
<body>
<header>
  <img src="/images/bu-logo.gif" alt="logo">
  <p>
  <center><b>{{title}}</b></center>
</header>

  {{{body}}}

</body>
</html>
```

The initial list of courses is now displayed in the browser as shown below.



The JavaScript file included in the web page has the actions for handling the button clicks as shown below.

clickActions.js

```
function addCourse(){
    window.location.href = '/courses/add';
}

function cancelAdd(){
    window.location.href = '/courses';
}
```

The *addCourse* module exports the function that renders the view for adding a new course.

addCourse.js

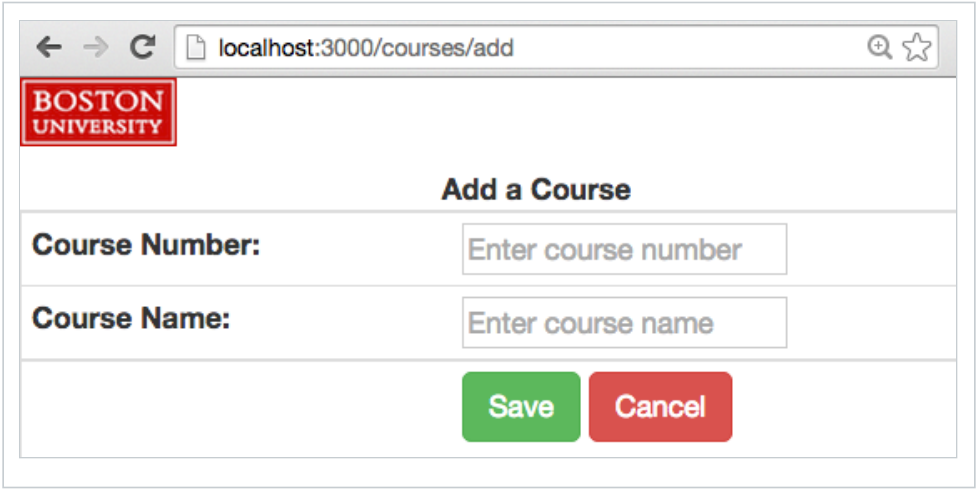
```
module.exports =
  function addCourse(req , res , next){
    res.render('addCourseView',
      {title:"Add a Course"});
};
```

The view page for adding a new course is shown below. In addition to the inputs for the course name and course number, buttons are provided for saving the course data or cancel the operation.

addCourseView.handlebars

```
<div class="table-responsive">
 <form method="post" action="/courses/add">
   <table class="table table-condensed">
     <tbody>
      <tr>
       <th>Course Number: </th>
       <td><input type="text" name="cnumber" required
            placeholder="Enter course number"></td>
      </tr>
      <tr>
        <th>Course Name: </th>
        <td><input type="text" name="cname" required
            placeholder="Enter course name"></td>
      </tr>
     </tbody>
     <tfooter>
       <tr>
         <td></td>
         <td>
          <input type="submit" value="Save"
            class="btn btn-success">
          <input type="button" value="Cancel"
            class="btn btn-danger" onclick="cancelAdd()">
         </td>
       </tr>
     </tfooter>
   </table>
 </form>
</div>
```

The initial layout of the view is shown below.

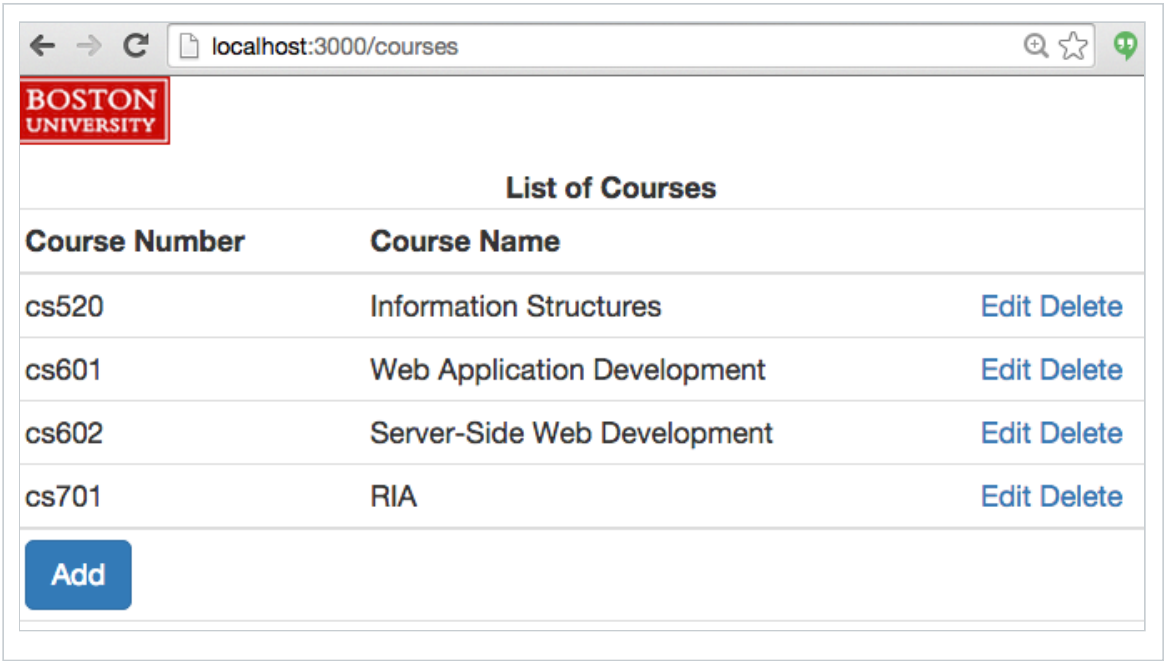


When the form is submitted, the *saveCourse* module exports the functionality for saving the data into the database. Since the form is submitted via POST, the course number and course name are obtained through the request's body.

saveCourse.js

```
var connection =
  require('./dbConnection.js').dbConnect();

module.exports =
  function saveCourse(req , res , next){

    var inputFromForm = {
      course_number  : req.body.cnumber,
      course_name    : req.body.cname
    };
    connection.query("INSERT INTO met_courses set ? ",
      inputFromForm,
      function(err, rows)
      {
        if (err)
          console.log("Error inserting : %s ",err );
        res.redirect('/courses');
      });
  };
```

After the save is successful, the application redirects the user to the course list as shown below. The last row in the list shows the information about the newly added course.



The *editCourse* module exports the function that renders the view for editing an existing course with the specified *id* through the request's parameter. The first row of the retrieved data is passed to the view page as shown below.

editCourse.js

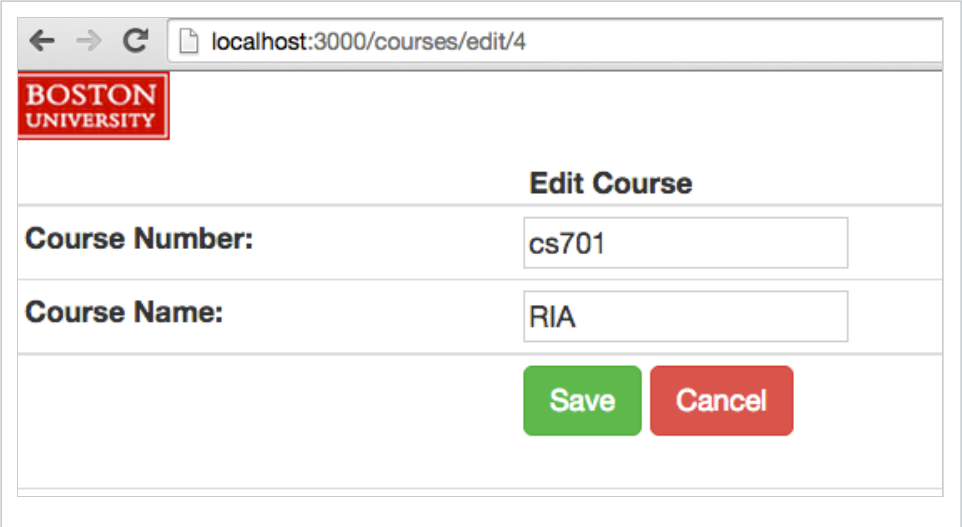
```
var connection =
  require('./dbConnection.js').dbConnect();

module.exports =
  function editCourse(req , res , next){
    var id = req.params.id;
    connection.query('SELECT * FROM met_courses WHERE id = ?',
      [id],
      function(err,rows){
        if(err)
          console.log("Error Selecting : %s ", err);
        res.render('editCourseView',
          {title:"Edit Course", data:rows[0]});
    });
};
```

The view page for editing the selected course is shown below. In addition to the inputs for the course name and course number, buttons are provided for saving the course data or cancel the operation.

editCourseView.handlebars

```
<div class="table-responsive">
  <form method="post" action='/courses/edit/{{data.id}}'>
    <table class="table table-condensed">
      <tbody>
        <tr>
          <th>Course Number: </td>
          <td><input type="text" name="cnumber" required
                value="{{data.course_number}}"></td>
        </tr>
        <tr>
          <th>Course Name: </td>
          <td><input type="text" name="cname" required
                value="{{data.course_name}}"></td>
        </tr>
      </tbody>
      <tfooter>
        <tr>
          <td></td>
          <td>
              <input type="submit" value="Save" class="btn btn-success">
              <input type="button" value="Cancel" class="btn btn-danger"
                class="cancel" onclick="cancelAdd()">
          </td>
        </tr>
      </tfooter>
    </table>
  </form>
</div>
```

The layout for the web page while editing the course is shown below. The existing values are shown in the input fields.

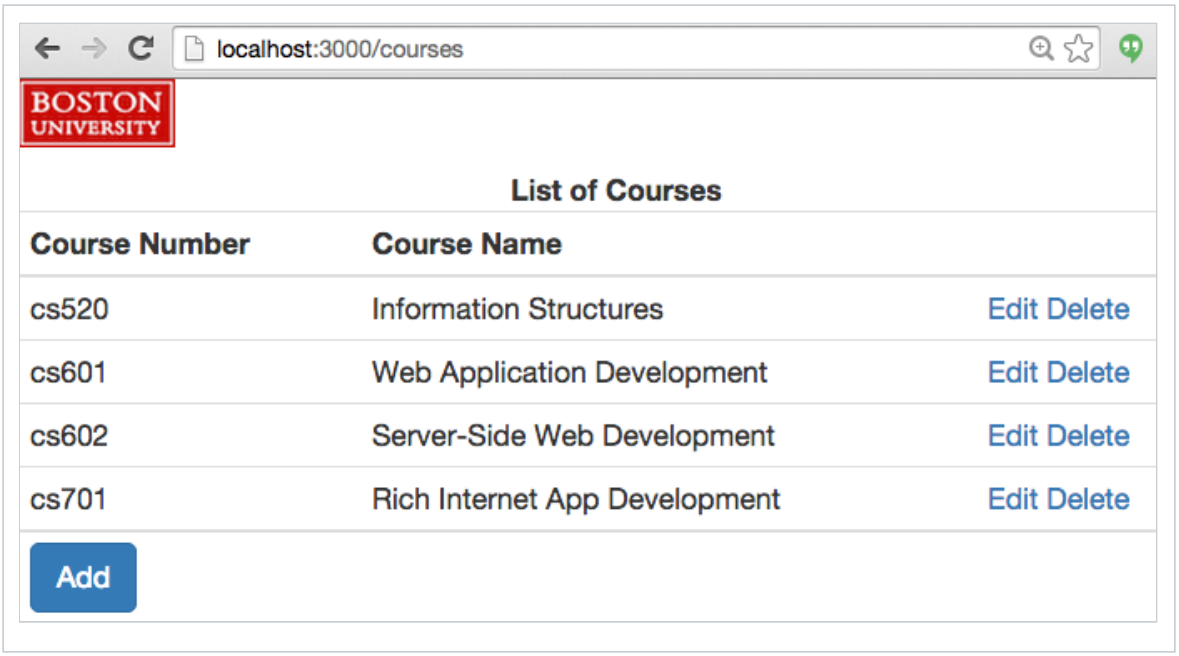


When the user submits this form for the modifications, the *saveAfterEdit* module exports the required functionality as shown below. The course number and course name are updated for the specified *id*. The former values are obtained through the request's body while the latter is retrieved from the request's parameters.

saveAfterEdit.js

```
1  var connection =
2    require('./dbConnection.js').dbConnect();
3
4  module.exports =
5    function saveCourse(req , res , next){
6      var id = req.params.id;
7      var inputFromForm = {
8        course_number  : req.body.cnumber,
9        course_name    : req.body.cname
10     };
11
12     connection.query("UPDATE met_courses set ? WHERE id=?",
13       [inputFromForm, id],
14       function(err, rows) {
15         if (err)
16             console.log("Error inserting : %s ",err );
17         res.redirect('/courses');
18     });
19   };
```

After the update is successful, the application redirects the user to the course list as shown below.

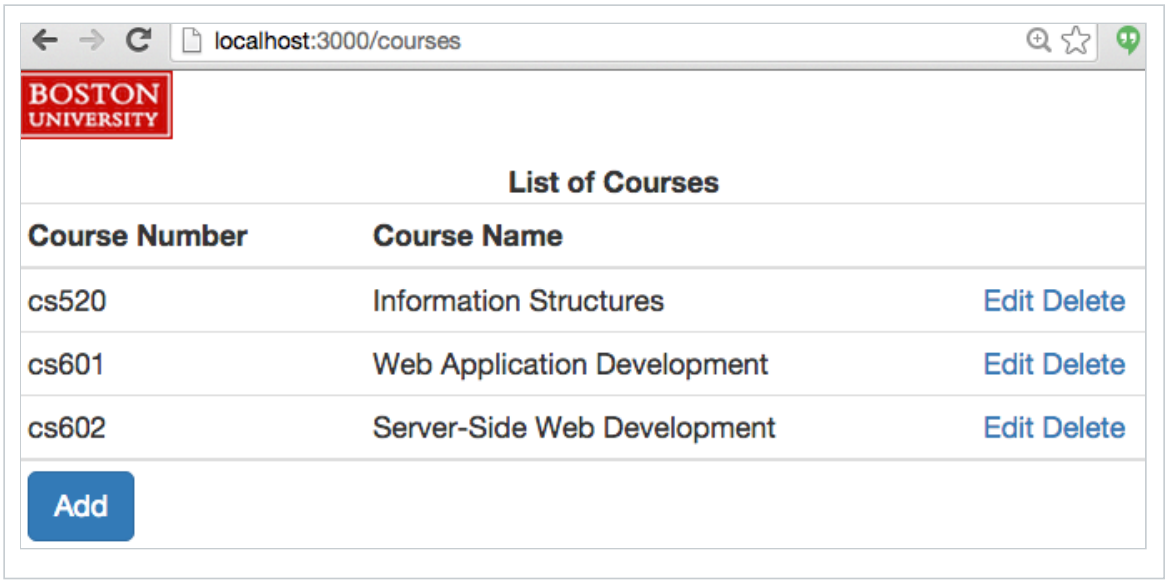


The *deleteCourse* module exports the function that deletes the specified course whose *id* is specified in the request's parameters.

deleteCourse.js

```
var connection =
  require('./dbConnection.js').dbConnect();

module.exports =
  function deleteCourse(req , res , next){
    var id = req.params.id;
    connection.query("DELETE FROM met_courses WHERE id = ? ",
      [id],
      function(err, rows)
      {
        if(err)
            console.log("Error deleting : %s ", err );
        res.redirect('/courses');
      });
  };
```

After the delete is successful, the application redirects the user to the course list as shown below.



References

Node MySQL, https://github.com/felixge/node-mysql

Persistence using MongoDB

Objectives

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

- Write standalone MySQL database applications
- Develop Node web applications that persist and interact with data form MySQL database

Introduction

MongoDB is one of the popular open source NoSQL databases used along with Node.js. MongoDB is a document-oriented database and stores data in BSON (Binary JSON) format. The instructions for installing MongoDB can be found on the web site:

https://www.mongodb.org/downloads

After installing MongoDB, run the mongod process located in the *bin* folder of the installation.

For the *Node* applications to interact with the MongoDB database, the *mongodb* module is required and is installed for the applications as shown below. This is the *Node.js* driver for MongoDB written in JavaScript.

```
>npm install --save mongodb
```

Connecting to MongoDB

A connection is required for accessing the MongoDB database. The connection is created using the connect() method from the *mongodb* module's *MongoClient* property. The first argument for this method is the connection string that specifies the database, *cs602db,* to be used in MongoDB. The second argument is the callback function invoked after the connection is successful. The second argument of the callback function gives access to the database object if there is no error. The database is created on the fly in MongoDB if it doesn't exist already.

The following example attempts the connection to the database, prints the name of the database, and closes the connection.

ex01_mongodb.js

```
var MongoClient = require('mongodb').MongoClient;

var dbUrl = 'mongodb://127.0.0.1:27017/cs602db';

MongoClient.connect(dbUrl, function (err, db) {
    if (err) throw err;
    console.log('Successfully connected to',
      db.s.databaseName);

    // Do database operations

    db.close();
});
```

The output of the above program on successful connection is shown below.

```
>node ex01_mongodb.js
Successfully connected to cs602db
```

If the MongoDB instance is not running, the above program results in an error while trying to establish a connection, as shown below.

```
Error: connect ECONNREFUSED
    at exports._errnoException (util.js:746:11)
    at TCPConnectWrap.afterConnect [as oncomplete] (net.js:1010:19)
```

Document Collection

A document is the unit of storage in the MongoDB database. A document is analogous to a record in the relational database. A collection in MongoDB is associated with a store of the documents for that collection. A collection is analogous to a table in the relational database. Since MongoDB is a schema free database, documents of various structures may be stored in the same collection. A collection, if it doesn't exist, is automatically created once a document is inserted.

Inserting a Document

The insert() method adds a document to the specified collection. If the document doesn't contain the *_id* field, MongoDB will create the *_id* property for each document object and assigns a unique *ObjectId* before inserting the document into the collection. If the *_id* field already exists in the document, the value has to be unique in the collection in order to avoid a duplicate key error.

The following example accesses the *people* document collection and inserts a new object that has *firstName* and *lastName* as properties. On successful insertion, the callback function's second argument can be used to examine the number of inserted documents, and their *_id* values.

ex02_mongodb.js

```
1  var MongoClient = require('mongodb').MongoClient;
2
3  var dbUrl = 'mongodb://127.0.0.1:27017/cs602db';
4
5  var newData = { firstName: 'John', lastName: 'Doe' };
6
7  MongoClient.connect(dbUrl, function (err, db) {
8    if (err) throw err;
9    console.log('Successfully connected');
10
11   var collection = db.collection('people');
12   collection.insert(newData, function (err, docs) {
13     console.log('Inserted Count:', docs.insertedCount);
14     console.log('Inserted Ids:', docs.insertedIds);
15     console.log('Data ID:', newData);
16
17     db.close();
18   });
19 });
```

The output of the above program is shown below. The *newData* object now has the *_id* property in addition to the *firstName* and *lastName*, upon insertion.

```
>node ex02_mongodb.js
Successfully connected
Inserted Count: 1
Inserted Ids: [ 55ceb89240d69ba146a2236b ]
Data ID: { firstName: 'John',
  lastName: 'Doe',
  _id: 55ceb89240d69ba146a2236b }
```

Finding all Documents

The find() method selects documents in the specified collection. The method can take two parameters, both of which are optional. The first optional argument is the query specifying the selection criteria. For returning all documents in a collection, the empty document {} is passed as the first argument, or can be omitted completely. The second optional argument specifies the fields to be returned in the matching documents. If this argument is omitted, all the fields are included in the results. The method returns a cursor to the matching documents that can converted to an array as shown in the example below. The callback function's second argument receives the array of matched documents.

ex03_mongodb.js

```
var MongoClient = require('mongodb').MongoClient;

var dbUrl = 'mongodb://127.0.0.1:27017/cs602db';

MongoClient.connect(dbUrl, function (err, db) {
  if (err) throw err;
  console.log('Successfully connected');

  var collection = db.collection('people');
  collection.find({}).toArray(function (err, docs) {
    console.log(docs);
    db.close();
  });
});
```

The output of the above program is shown below.

```
>node ex03_mongodb.js
Successfully connected
[ { _id: 55ceb89240d69ba146a2236b,
    firstName: 'John',
    lastName: 'Doe' } ]
```

Inserting Multiple Documents

The insert() method can also be used with a single call to add multiple documents to the specified collection. The following example shows the array of objects with *firstName* and *lastName* properties to be inserted into the *people* collection. On successful insertion, the callback function's second argument can be used to examine the number of inserted documents, and their *_id* values.

ex04_mongodb.js

```
1  var MongoClient = require('mongodb').MongoClient;
2
3  var dbUrl = 'mongodb://127.0.0.1:27017/cs602db';
4
5  var newData = [
6    { firstName: 'Jane', lastName: 'Doe' },
7    { firstName: 'John', lastName: 'Smith' },
8    { firstName: 'Josh', lastName: 'Smith' }];
9
10 MongoClient.connect(dbUrl, function (err, db) {
11   if (err) throw err;
12   console.log('Successfully connected');
13
14   var collection = db.collection('people');
15   collection.insert(newData, function (err, docs) {
16     console.log('Inserted Count:', docs.insertedCount);
17     console.log('Inserted Ids:', docs.insertedIds);
18
19     db.close();
20   });
21 });
```

The output of the above program is shown below, illustrating the successful insertion of the three documents.

```
>node ex04_mongodb.js
Successfully connected
Inserted Count: 3
Inserted Ids: [ 55ceb903442545b146a25d34,
  55ceb903442545b146a25d35,
  55ceb903442545b146a25d36 ]
```

The output of the program finding all the documents in the *people* collection is shown below with the four documents that were inserted so far.

```
>node ex03_mongodb.js
Successfully connected
[ { _id: 55ceb89240d69ba146a2236b,
    firstName: 'John',
    lastName: 'Doe' },
  { _id: 55ceb903442545b146a25d34,
    firstName: 'Jane',
    lastName: 'Doe' },
  { _id: 55ceb903442545b146a25d35,
    firstName: 'John',
    lastName: 'Smith' },
  { _id: 55ceb903442545b146a25d36,
    firstName: 'Josh',
    lastName: 'Smith' } ]
```

Finding Documents with Query Criteria

The following example shows the usage of the find() method for selecting documents in the collection matching the specified query criteria. All documents with the given *lastName,* or the given *firstName*, or both the *firstName* and *lastName* can be selected as shown below.

ex05_mongodb.js

```
1  var MongoClient = require('mongodb').MongoClient;
2
3  var dbUrl = 'mongodb://127.0.0.1:27017/cs602db';
4
5  MongoClient.connect(dbUrl, function (err, db) {
6    if (err) throw err;
7    console.log('Successfully connected');
8
9    var collection = db.collection('people');
10   collection.find({lastName: 'Smith'}).toArray(
11     function (err, docs) {
12       console.log("Search by lastName:\n", docs);
13   });
14
15   collection.find({firstName: 'John'}).toArray(
16     function (err, docs) {
17       console.log("Search by firstName:\n", docs);
18   });
19
20   collection.find({firstName: 'John', lastName: 'Smith'}).toArray(
21     function (err, docs) {
22       console.log("Search by first and last Name:\n", docs);
23       db.close();
24   });
25 });
```

The output of the above program is shown below. From the document collection, there are two people with the last name *Smith*, two people with the first name *John*, and one person with the name *John Smith*.

```
>node ex05_mongodb.js
Successfully connected
Search by lastName:
 [ { _id: 55ceb903442545b146a25d35,
    firstName: 'John',
    lastName: 'Smith' },
  { _id: 55ceb903442545b146a25d36,
    firstName: 'Josh',
    lastName: 'Smith' } ]
Search by firstName:
 [ { _id: 55ceb89240d69ba146a2236b,
    firstName: 'John',
    lastName: 'Doe' },
  { _id: 55ceb903442545b146a25d35,
    firstName: 'John',
    lastName: 'Smith' } ]
Search by first and last Name:
 [ { _id: 55ceb903442545b146a25d35,
    firstName: 'John',
    lastName: 'Smith' } ]
```

Updating Documents

The update() method is used for modifying a document or multiple documents in the specified collection. The method can modify specific fields of existing document or documents, or replace an existing document entirely. By default, the method updates a single document. The first argument is the query specifying the selection criteria for the update. The second argument is the update criteria.

The following example updates the document with the specified *firstName* and *lastName* and changes the *firstName* of the document from *Josh* to *James*.

ex06_mongodb.js

```
var MongoClient = require('mongodb').MongoClient;

var dbUrl = 'mongodb://127.0.0.1:27017/cs602db';

MongoClient.connect(dbUrl, function (err, db) {
  if (err) throw err;
  console.log('Successfully connected');

  var collection = db.collection('people');

  collection.update({firstName: 'Josh', lastName: 'Smith'},
      {$set: {firstName: 'James'}},
    function (err, docs) {
      console.log(docs.result);
      db.close();
  });
});
```

The output of the above program is shown below. On successful update, the callback function's second argument can be used to see the number of documents that were modified.

```
>node ex06_mongodb.js
Successfully connected
{ ok: 1, nModified: 1, n: 1 }
```

The output of the program finding all the documents in the *people* collection is shown below.

```
>node ex03_mongodb.js
Successfully connected
[ { _id: 55ceb89240d69ba146a2236b,
    firstName: 'John',
    lastName: 'Doe' },
  { _id: 55ceb903442545b146a25d34,
    firstName: 'Jane',
    lastName: 'Doe' },
  { _id: 55ceb903442545b146a25d35,
    firstName: 'John',
    lastName: 'Smith' },
  { _id: 55ceb903442545b146a25d36,
    firstName: 'James',
    lastName: 'Smith' } ]
```

The following example shows the update by querying the *lastName*. Even though the document collection has two documents with the specified last name, only the first matched document is updated by default.

ex07_mongodb.js

```
var MongoClient = require('mongodb').MongoClient;

var dbUrl = 'mongodb://127.0.0.1:27017/cs602db';

MongoClient.connect(dbUrl, function (err, db) {
  if (err) throw err;
  console.log('Successfully connected');

  var collection = db.collection('people');

  collection.update({lastName: 'Doe'},
      {$set: {lastName: 'Harris'}},
    function (err, docs) {
      console.log(docs.result);
      db.close();
  });
});
```

The result of the above program is shown below illustrating that only one document is updated.

```
>node ex07_mongodb.js
Successfully connected
{ ok: 1, nModified: 1, n: 1 }
```

The output of the program finding all the documents in the *people* collection is shown below.

```
>node ex03_mongodb.js
Successfully connected
[ { _id: 55ceb89240d69ba146a2236b,
    firstName: 'John',
    lastName: 'Harris' },
  { _id: 55ceb903442545b146a25d34,
    firstName: 'Jane',
    lastName: 'Doe' },
  { _id: 55ceb903442545b146a25d35,
    firstName: 'John',
    lastName: 'Smith' },
  { _id: 55ceb903442545b146a25d36,
    firstName: 'James',
    lastName: 'Smith' } ]
```

For updating multiple matched documents at the same time, the optional third argument {multi: true} may be specified. If the {upsert: true} option is specified, the new document is inserted in the collection if there is no document that matches the specified query.

Deleting Documents

The remove() method is used for deleting a document or multiple documents in the specified collection. The first argument for this method is the query for selecting the documents to be removed. Specifying the empty document {} removes all documents from the collection. An optional second argument {justOne: true} can be specified to restrict the deletion to a single document.

The following example removes all documents with the specified *lastName*.

ex08_mongodb.js

```
var MongoClient = require('mongodb').MongoClient;

var dbUrl = 'mongodb://127.0.0.1:27017/cs602db';

MongoClient.connect(dbUrl, function (err, db) {
  if (err) throw err;
  console.log('Successfully connected');

  var collection = db.collection('people');
  collection.remove({lastName: 'Smith'},
    function (err, results) {
      console.log(results.result);
      db.close();
  });
});
```

The result of the above program is shown below illustrating that two documents are deleted.

```
>node ex08_mongodb.js
Successfully connected
{ ok: 1, n: 2 }
```

The output of the program finding all the documents in the *people* collection is shown below.

```
>node ex03_mongodb.js
Successfully connected
[ { _id: 55ceb89240d69ba146a2236b,
    firstName: 'John',
    lastName: 'Harris' },
  { _id: 55ceb903442545b146a25d34,
    firstName: 'Jane',
    lastName: 'Doe' } ]
```

Mongoose ODM

MongoDB works with simple JSON documents. However, the business logic for the application working with these documents must be written elsewhere. An Object Document Mapper encapsulates both the data and the methods for validation and business logic. Mongoose is one of the popular ODMs officially supported by MongoDB. The *mongoose* module is first installed using *npm* as shown below.

```
>npm install --save mongoose
```

Mongoose Schema and Model

Mongoose provides the *Schema* class for defining all the fields of the document along with their types. After the *Schema* is defined, custom methods can also be defined that can be used on the model instances. The *model* function is used for defining the model associated with the schema. The *model* provides the constructor for creating instances of the document. Custom methods can then be invoked on these instances as shown in the example below.

test.js

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

// Define the schema
var personSchema = new Schema({
  firstName: 'string',
  lastName: 'string'
});

// Optionally, add custom methods
personSchema.methods.printMe =
  function() {
    console.log("I am", this.firstName +
      ' ' + this.lastName);
  };

// Create a Model
var Person = mongoose.model('PersonModel',
      personSchema);

// create an instance
var john = new Person({
  firstName: 'John', lastName: 'Smith'
});
// invoke custom method
john.printMe();
```

The following examples use the *Schema* for capturing the course details: *courseNumber*, *courseName*, and *courseDevelopers*. The first two properties for each course document are of the *string* type. The *courseDevelopers* property is an array of objects, each with the *firstName* and *lastName* properties. After the *Schema* is defined, the module exports the function *getModel* that creates the model for the schema using the MongoDB database connection object. Based on the given model name in the example, the documents will be created in the collection named *coursemodels.*

coursesDb.js

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var courseSchema = new Schema({
  courseNumber: String,
  courseName: String,
  courseDevelopers: [
    {firstName: String, lastName: String}
  ]
});

module.exports = {
  getModel: function getModel(connection) {
    return connection.model("CourseModel",
                  courseSchema);
  }
}
```

Mongoose – Creating Documents

The sample application loads the *mongoose* module and creates the MongoDB connection object. The schema module is loaded and the model object is created using this database connection.

ex09_mongoose.js

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

var dbUrl = 'mongodb://127.0.0.1:27017/cs602db';
var connection = mongoose.createConnection(dbUrl);

var CourseDb = require('./coursesDb.js');
var Course = CourseDb.getModel(connection);

```

When the database connection is established, the *Course* objects can be created and saved to the database using the save() method as shown below. After the last course is saved, the connection is closed in the callback function after the save is done.

```
connection.on("open", function(){

  // create and save document objects
  var course;

  course = new Course({...
  });
  course.save();

  course = new Course({...
  });
  course.save();

  course = new Course({...
  });
  course.save(function(err) {
    connection.close();
    if (err) throw err;
    console.log("Success!");
  });

});
```

The properties used for the above course document objects are shown below. The first and last courses have one course developer, while the middle course has two course developers.

```
14  course = new Course({
15    courseNumber: 'cs601',
16    courseName: 'Web Application Development',
17    courseDevelopers: [
18      { firstName: 'Eric',lastName: 'Bishop'}
19    ]
20  });
21  course.save();
22
23  course = new Course({
24    courseNumber: 'cs602',
25    courseName: 'Server Side Web Development',
26    courseDevelopers: [
27      { firstName: 'Eric', lastName: 'Bishop'},
28      { firstName: 'Suresh',lastName: 'Kalathur'}
29    ]
30  });
31  course.save();
32
33  course = new Course({
34    courseNumber: 'cs701',
35    courseName: 'RIA',
36    courseDevelopers: [
37      { firstName: 'Suresh',lastName: 'Kalathur'}
38    ]
39  });
40  course.save(function(err) {
41    connection.close();
42    if (err) throw err;
43    console.log("Success!");
44  });
```

Executing the above program successfully inserts the documents into the MongoDB database.

```
>node ex09_mongoose.js
Success!
```

Mongoose – Finding Documents

The *mongo* terminal client can be used to inspect the data from the *coursemodels* collection as shown below. The find() method returns all the three documents. Note that the course developer objects are embedded within each document.

```
mongo> db.coursemodels.find({})
{ "_id" : ObjectId("55d0f0c2a931eed64f18e0e8"), "courseNumber" : "cs601",
 "courseName" : "Web Application Development", "courseDevelopers" : [ { "
firstName" : "Eric", "lastName" : "Bishop", "_id" : ObjectId("55d0f0c2a93
1eed64f18e0e9") } ], "__v" : 0 }
{ "_id" : ObjectId("55d0f0c2a931eed64f18e0ed"), "courseNumber" : "cs701",
 "courseName" : "RIA", "courseDevelopers" : [ { "firstName" : "Suresh", "
lastName" : "Kalathur", "_id" : ObjectId("55d0f0c2a931eed64f18e0ee") } ],
 "__v" : 0 }
{ "_id" : ObjectId("55d0f0c2a931eed64f18e0ea"), "courseNumber" : "cs602",
 "courseName" : "Server Side Web Development", "courseDevelopers" : [ { "
firstName" : "Eric", "lastName" : "Bishop", "_id" : ObjectId("55d0f0c2a93
1eed64f18e0ec") }, { "firstName" : "Suresh", "lastName" : "Kalathur", "_i
d" : ObjectId("55d0f0c2a931eed64f18e0eb") } ], "__v" : 0 }
```

The find() method can also be used on the model object for retrieving the documents matching the specified query. In the following example, the results are projected showing only the *courseName* and *courseNumber* properties (*_id* is included by default).

ex10_mongoose.js

```
10
11    // Find all courses
12    Course.find({}, 'courseName courseNumber',
13      function(err, results) {
14        if (err) throw err;
15        console.log("\nFind all courses");
16        console.log(results);
17    });
```

The output of the above program is shown below.

```
Find all courses
[ { _id: 55d0f0c2a931eed64f18e0e8,
    courseNumber: 'cs601',
    courseName: 'Web Application Development' },
  { _id: 55d0f0c2a931eed64f18e0ed,
    courseNumber: 'cs701',
    courseName: 'RIA' },
  { _id: 55d0f0c2a931eed64f18e0ea,
    courseNumber: 'cs602',
    courseName: 'Server Side Web Development' } ]
```

The following example retrieves the entire document matching the *courseNumber* property with the specified value.

ex10_mongoose.js

```
19
20   Course.find({courseNumber: 'cs602'},
21     function(err, results) {
22       if (err) throw err;
23       console.log("\nFind cs602");
24       console.log(results);
25   });
```

The output of the above program is shown below.

```
Find cs602
[ { _id: 55d0f0c2a931eed64f18e0ea,
    courseNumber: 'cs602',
    courseName: 'Server Side Web Development',
    __v: 0,
    courseDevelopers:
     [ { firstName: 'Eric',
         lastName: 'Bishop',
         _id: 55d0f0c2a931eed64f18e0ec },
       { firstName: 'Suresh',
         lastName: 'Kalathur',
         _id: 55d0f0c2a931eed64f18e0eb } ] } ]
```

The following example imposes the restriction on the *courseDevelopers* whose size is not equal to 1. The query matches all documents in the collection that either doesn't have the property, or has the property with size not equal to 1.

ex10_mongoose.js

```
26
27   Course.find({courseDevelopers: {$not: {$size: 1}}},
28     function(err, results) {
29       connection.close();
30       if (err) throw err;
31       console.log("\nCourses with multiple develeopers");
32       console.log(results);
33   });
```

The output of the program shows only one course document among the three that has two course developers.

```
Courses with multiple develeopers
[ { _id: 55d0f0c2a931eed64f18e0ea,
    courseNumber: 'cs602',
    courseName: 'Server Side Web Development',
    __v: 0,
    courseDevelopers:
     [ { firstName: 'Eric',
         lastName: 'Bishop',
         _id: 55d0f0c2a931eed64f18e0ec },
       { firstName: 'Suresh',
         lastName: 'Kalathur',
         _id: 55d0f0c2a931eed64f18e0eb } ] } ]
```

Mongoose Model – Updating Documents

The model's update() method is used for updating the properties of the document that matches the query provided by the first argument. The following example updates the *courseName* of the document with the specified *courseNumber*.

ex11_mongoose.js

```
11  // Update
12  Course.update({courseNumber: 'cs701'},
13      {courseName: 'Rich Internet App Development'},
14    function(err, result) {
15      connection.close();
16      if (err) throw err;
17      console.log("\nUpdate...");
18      console.log("Affected docs:", result);
19  });
```

The output of the above program is shown below showing the number of documents modified .

```
>node ex11_mongoose.js

Update...
Affected docs: { ok: 1, nModified: 1, n: 1 }
```

The *mongo* terminal client can be used to inspect the data for the modified course as shown below.

```
mongo> db.coursemodels.find({courseNumber: 'cs701'})
{ "_id" : ObjectId("55d0f0c2a931eed64f18e0ed"), "courseNumber" : "cs701",
 "courseName" : "Rich Internet App Development", "courseDevelopers" : [ {
 "firstName" : "Suresh", "lastName" : "Kalathur", "_id" : ObjectId("55d0f
0c2a931eed64f18e0ee") } ], "__v" : 0 }
```

For updating multiple documents, the *multi* property is set to *true* as shown in the example below. Since the query ({}) matches all the documents in the model, all documents now will have the same *courseName* as specified.

ex11_mongoose.js

```
21    // Update multiple documents
22    Course.update({},
23        {courseName: 'Web Development Course'},
24        {multi: true},
25      function(err, result) {
26        connection.close();
27        if (err) throw err;
28        console.log("\nUpdate...");
29        console.log("Affected docs:", result);
30    });
31  });
```

The output of the above program is shown below. The three documents in the collection are modified as the result of the operation.

```
>node ex11_mongoose.js

Update...
Affected docs: { ok: 1, nModified: 3, n: 3 }
```

The *mongo* terminal client can be used to inspect the data for all the courses as shown below.

```
mongo> db.coursemodels.find({}, {courseNumber: '1', courseName: '1'})
{ "_id" : ObjectId("55d0f0c2a931eed64f18e0e8"), "courseNumber" : "cs601",
 "courseName" : "Web Development Course" }
{ "_id" : ObjectId("55d0f0c2a931eed64f18e0ed"), "courseNumber" : "cs701",
 "courseName" : "Web Development Course" }
{ "_id" : ObjectId("55d0f0c2a931eed64f18e0ea"), "courseNumber" : "cs602",
 "courseName" : "Web Development Course" }
```

Mongoose Model – Deleting Documents

The model's remove() method is used for deleting the documents that match the query provided by the first argument. The following example deletes the document with the specified *courseNumber*.

ex12_mongoose.js

```
11    // Delete documents
12    Course.remove({courseNumber: 'cs602'},
13      function(err, result) {
14        connection.close();
15        if (err) throw err;
16        console.log("\Delete...");
17        console.log("Affected docs:", result);
18    });
```

The output of the above program is shown below. One document is deleted as a result.

```
>node ex12_mongoose.js
Delete...
Affected docs: { result: { ok: 1, n: 1 },
  connection:
   { domain: null,
     events: {}
```

The *mongo* terminal client can be used to inspect the data for the remaining courses as shown below.

```
mongo> db.coursemodels.find({}, {courseNumber: '1', courseName: '1'})
{ "_id" : ObjectId("55d0f0c2a931eed64f18e0e8"), "courseNumber" : "cs601",
 "courseName" : "Web Development Course" }
{ "_id" : ObjectId("55d0f0c2a931eed64f18e0ed"), "courseNumber" : "cs701",
 "courseName" : "Web Development Course" }
```

The following example used regular expression pattern to remove all course documents with *Web* included in the course name.

ex12_mongoose.js

```
20    Course.remove({courseName:
21          { "$regex": "Web", "$options": "i" }},
22      function(err, result) {
23        connection.close();
24        if (err) throw err;
25        console.log("\Delete...");
26        console.log("Affected docs:", result.result);
27    });
```

The result of the above program is shown below, deleting the remaining two documents.

```
>node ex12_mongoose.js
Delete...
Affected docs: { ok: 1, n: 2 }
```

The *mongo* terminal client can be used to inspect the data for the remaining documents (none are remaining).

```
mongo> db.coursemodels.find({})
mongo>
```

Case Study – Course Manager

The following example shows the web application to manage the list of courses using Node, Express, MongoDB, and Mongoose. The schema for the document database is shown below.

dbConnection.js

```
1  var mongoose = require('mongoose');
2
3  var dbUrl = 'mongodb://127.0.0.1:27017/cs602db';
4  var connection = null;
5  var model = null;
6
7  var Schema = mongoose.Schema;
8
9  var courseSchema = new Schema({
10   courseNumber: String,
11   courseName: String,
12   courseDevelopers: [
13     {firstName: String, lastName: String}
14   ]
15 });
16 // custom schema method
17 courseSchema.methods.getDeveloperNames =
18     function() {
19       return this.courseDevelopers.map(
20                 function (elem){
21                   return elem.firstName + ' ' +
22                          elem.lastName;
23                 }).join(',');
24     };
```

The schema also adds a custom method that returns a comma-separated string joining the first and last names of all the course developers for the given course. The module exports the function, *getModel*. The function checks if the connection to the database already exists. If not, the function creates the connection and creates the model with the defined schema.

dbConnection.js

```
25
26  module.exports = {
27    getModel: function getModel() {
28      if (connection == null) {
29        console.log("Creating connection and model...");
30        connection = mongoose.createConnection(dbUrl);
31        model = connection.model("CourseModel",
32                                 courseSchema);
33      };
34      return model;
35    }
36  };
```

The web application provides the capability for displaying all the courses, edit or delete an existing course, and adding a new course. The web application uses Express and Express Handlebars as the view template engine. The Express Router is used for specifying the end points and their handlers, as shown below.

ex13_caseStudy.js

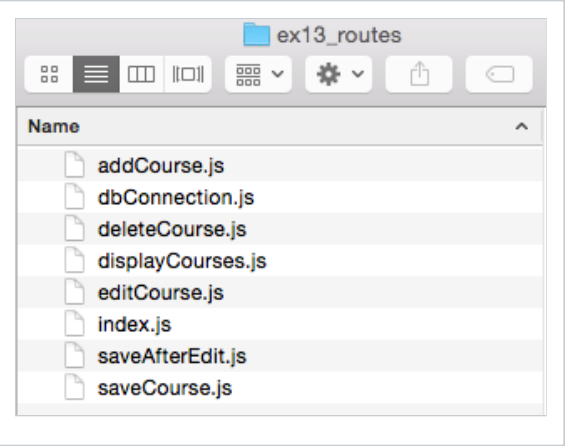
```
1  var express = require('express');
2  var bodyParser = require('body-parser');
3  var handlebars = require('express-handlebars');
4
5  var app = express();
6
7  // setup handlebars view engine
8  app.engine('handlebars',
9      handlebars({defaultLayout: 'main_logo'}));
10 app.set('view engine', 'handlebars');
11
12 // static resources
13 app.use(express.static(__dirname + '/public'));
14
15 app.use(bodyParser.json());
16 app.use(bodyParser.urlencoded({ extended: false }));
17
18 // Routing
19 var routes = require('./ex13_routes/index');
20 app.use('/', routes);
21
```

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

Route	HTTP	Function
/courses	GET	Display all the courses from the database
/courses/add	GET	Show form to add a new course

/courses/add	POST	Save the new course to the database
/courses/edit/:id	GET	Show form with the data about the course with the specified id
/courses/edit/:id	POST	Update the course data in the database for the specified id
/courses/delete/:id	GET	Delete the course with the specified id from the database

Each of the above operations is specified in their own modules as shown in the following structure.



The *index.js* file creates the router with the routes and their handlers as shown below. The module exported by this file is used in the main web application.

index.js

```
var express = require('express');
var router = express.Router();

// other modules
var displayCourses  = require("./displayCourses");
var addCourse       = require("./addCourse");
var saveCourse      = require("./saveCourse");
var editCourse      = require("./editCourse");
var saveAfterEdit   = require("./saveAfterEdit");
var deleteCourse    = require("./deleteCourse");

// router specs
router.get('/', function(req, res, next) {
  res.redirect('/courses');
});

router.get('/courses',             displayCourses);

router.get('/courses/add',         addCourse);
router.post('/courses/add',        saveCourse);

router.get('/courses/edit/:id',    editCourse);
router.post('/courses/edit/:id',   saveAfterEdit);

router.get('/courses/delete/:id', deleteCourse);

module.exports = router;
```

The *displayCourses* module exports the function that retrieves all the relevant documents from the MongoDB database, maps over the courses, and passes the customized data to the view as shown below.

displayCourses.js

```
var DB = require('./dbConnection.js');
var Course = DB.getModel();

module.exports =
  function displayCourses(req , res , next){
    Course.find({}, function(err , courses){
      if(err)
        console.log("Error : %s ",err);

      var results = courses.map(function (course){
        return {
          id: course._id,
          courseName: course.courseName,
          courseNumber: course.courseNumber,
          developers:  course.getDeveloperNames()
        }
      });
      res.render('displayCoursesView',
        {title:"List of Courses", data:results});
    });
};
```

The view page for displaying all the course data is shown below. In addition to the course name, course number and course developers, links are provided for editing and deleting each course. The view also has the button for adding a new course.

displayCoursesView.handlebars

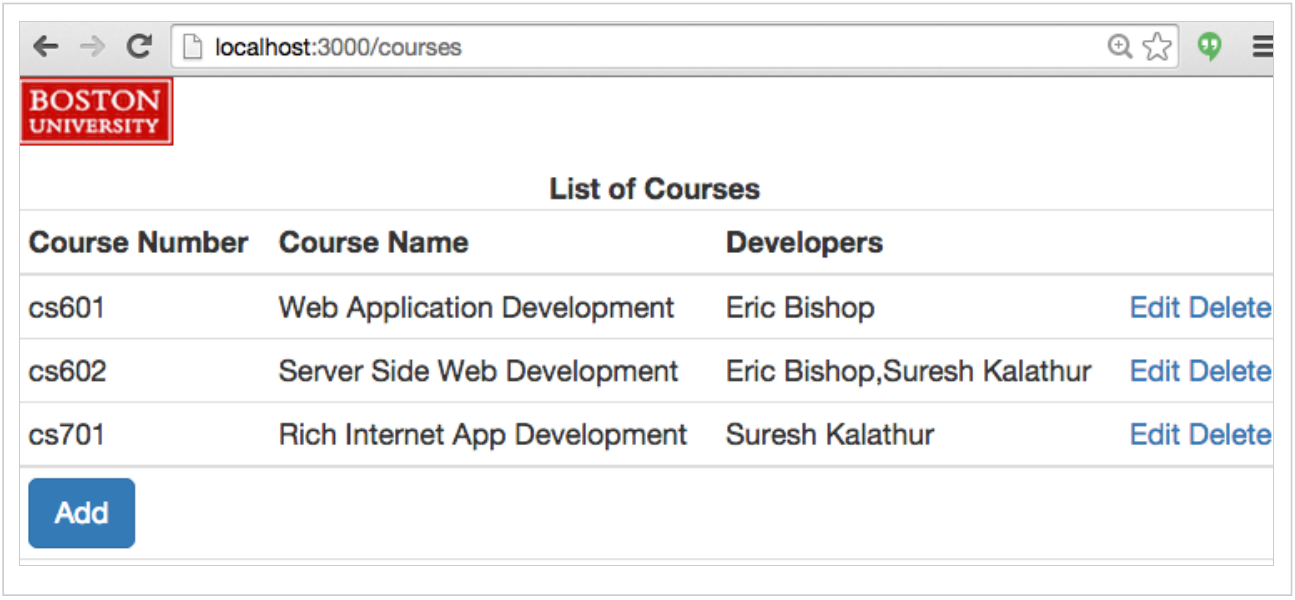
```
<div>
  <div class="table-responsive">
    <table class="table table-condensed">
      <thead>
        <tr>
          <th>Course Number</th>
          <th>Course Name</th>
          <th>Developers</th>
          <th></th>
        </tr>
      </thead>
      <tbody>
        {{#each data}}
          <tr>
            <td>{{this.courseNumber}}</td>
            <td>{{this.courseName}}</td>
            <td>{{this.developers}}</td>
            </td>
            <td>
              <a href="/courses/edit/{{this.id}}">Edit</a>
              <a href="/courses/delete/{{this.id}}">Delete</a>
            </td>
          </tr>
        {{/each}}
      </tbody>
      <tfooter>
        <tr><td>
        <button class="btn btn-primary"
            onClick="addCourse();">Add</button>
        </td></tr>
      </tfooter>
    </table>
  </div>
</div>
```

The following layout is used for all the views in the application.

main_logo.handlebars

```
<!doctype html>
<html>
<head>
  <title>CS602</title>
  <link rel='stylesheet' href='/stylesheets/bootstrap.css' />
  <script src="/javascripts/clickActions.js"
    type="text/javascript"></script>
</head>
<body>
<header>
  <img src="/images/bu-logo.gif" alt="logo">
  <p>
  <center><b>{{title}}</b></center>
</header>

  {{{body}}}

</body>
</html>
```

The initial list of courses is now displayed in the browser as shown below.



The JavaScript file included in the web page has the actions for handling the button clicks as shown below.

clickActions.js

```
function addCourse(){
    window.location.href = '/courses/add';
}

function cancelAdd(){
    window.location.href = '/courses';
}
```

The *addCourse* module exports the function that renders the view for adding a new course.

addCourse.js

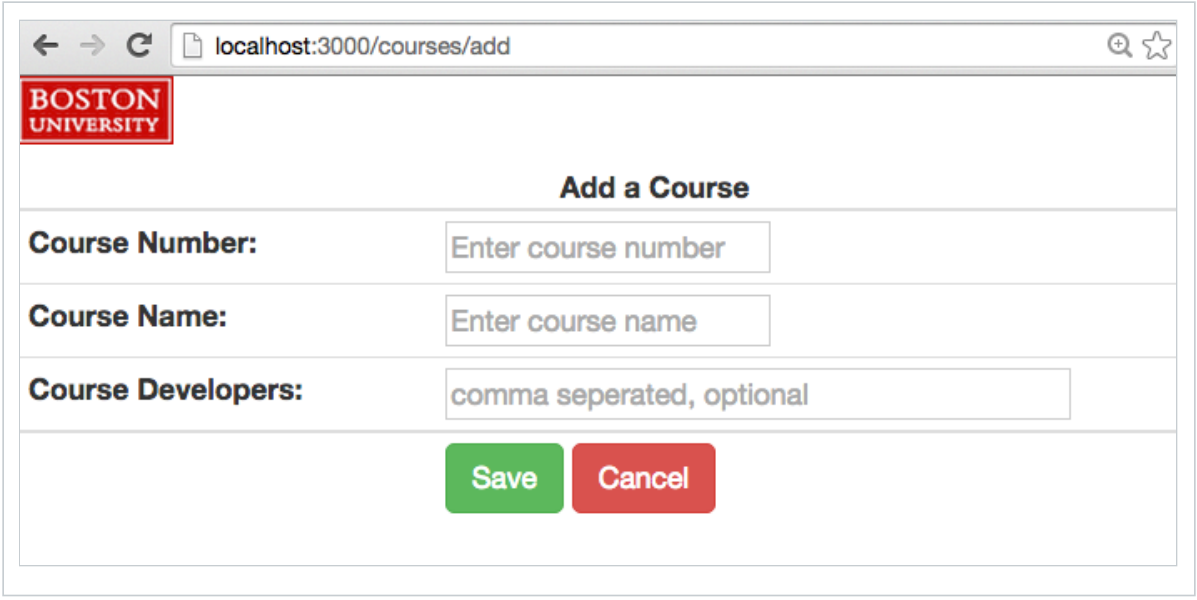
```
module.exports =
  function addCourse(req , res , next){
    res.render('addCourseView',
      {title:"Add a Course"});
};
```

The view page for adding a new course is shown below. In addition to the inputs for the course name, course number, and course developers, buttons are provided for saving the course data or cancel the operation.

addCourseView.handlebars

```
<div class="table-responsive">
 <form method="post" action="/courses/add">
   <table class="table table-condensed">
     <tbody>
      <tr>
       <th width="200">Course Number: </th>
       <td><input type="text" name="cnumber" required
            placeholder="Enter course number"></td>
      </tr>
      <tr>
        <th>Course Name: </th>
        <td><input type="text" name="cname" required
            placeholder="Enter course name"></td>
      </tr>
      <tr>
        <th>Course Developers: </th>
        <td><input type="text" name="cdev" size="40"
            placeholder="comma seperated, optional"></td>
      </tr>
     </tbody>
     <tfooter>
       <tr>
         <td></td>
         <td>
          <input type="submit" value="Save"
            class="btn btn-success">
          <input type="button" value="Cancel"
            class="btn btn-danger" onclick="cancelAdd()">
         </td>
       </tr>
     </tfooter>
   </table>
 </form>
</div>
```

The initial layout of the view is shown below.

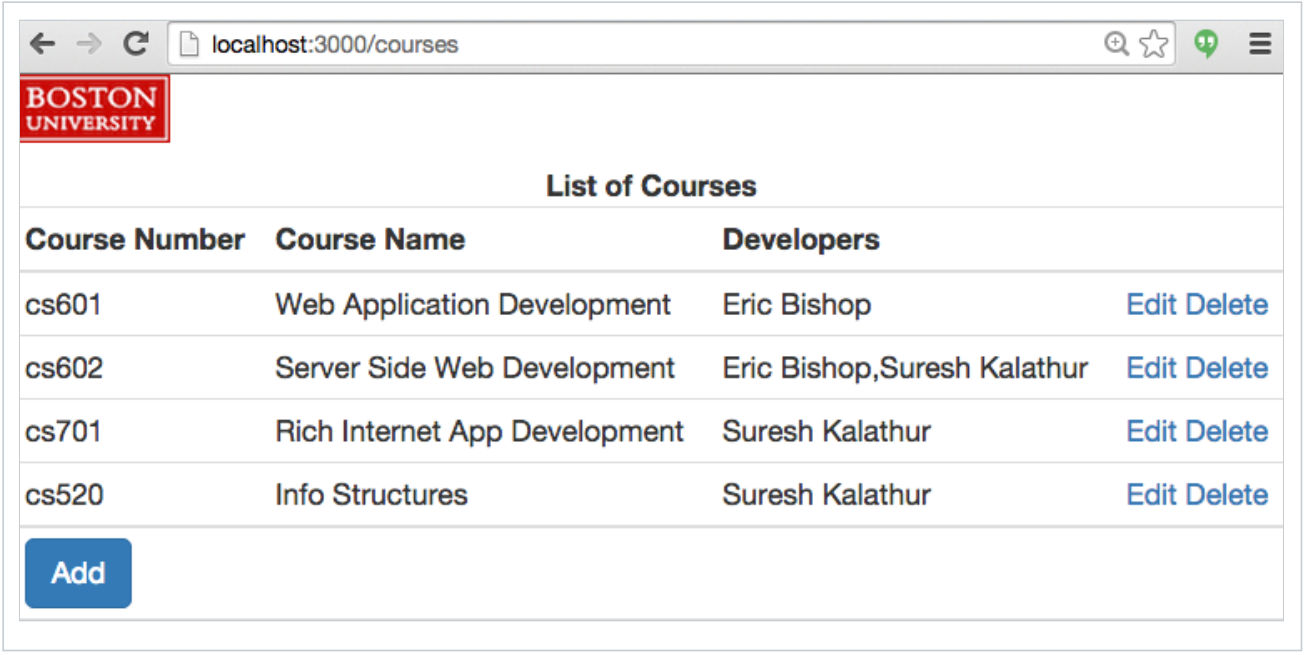


When the form is submitted, the *saveCourse* module exports the functionality for saving the data into the database. Since the form is submitted via POST, the course number, course name, and course developers are obtained through the request's body. The comma-separated data of course developers is parsed and an object with the *firstName* and *lastName* properties is created for each developer.

saveCourse.js

```
var DB = require('./dbConnection.js');
var Course = DB.getModel();

module.exports =
  function saveCourse(req , res , next){

    var developers = req.body.cdev;
    // create an array of objects
    if (developers.length > 0) {
      developers =
        developers.split(',').map(function (elem){
          var names = elem.trim().split(' ');
          return {firstName: names[0],
                  lastName: names[1]};
        });
    } else
      developers = [];

    var course = new Course({
      courseNumber:     req.body.cnumber,
      courseName:       req.body.cname,
      courseDevelopers: developers
    });

    course.save(function (err){
      if(err)
        console.log("Error : %s ",err);
      res.redirect('/courses');
    });

  };
```

After the save is successful, the application redirects the user to the course list as shown below. The last row in the list shows the information about the newly added course.



The *editCourse* module exports the function that renders the view for editing an existing course with the specified *id* through the request's parameter. A new object is created from the course data and is passed to the view page as shown below.

editCourse.js

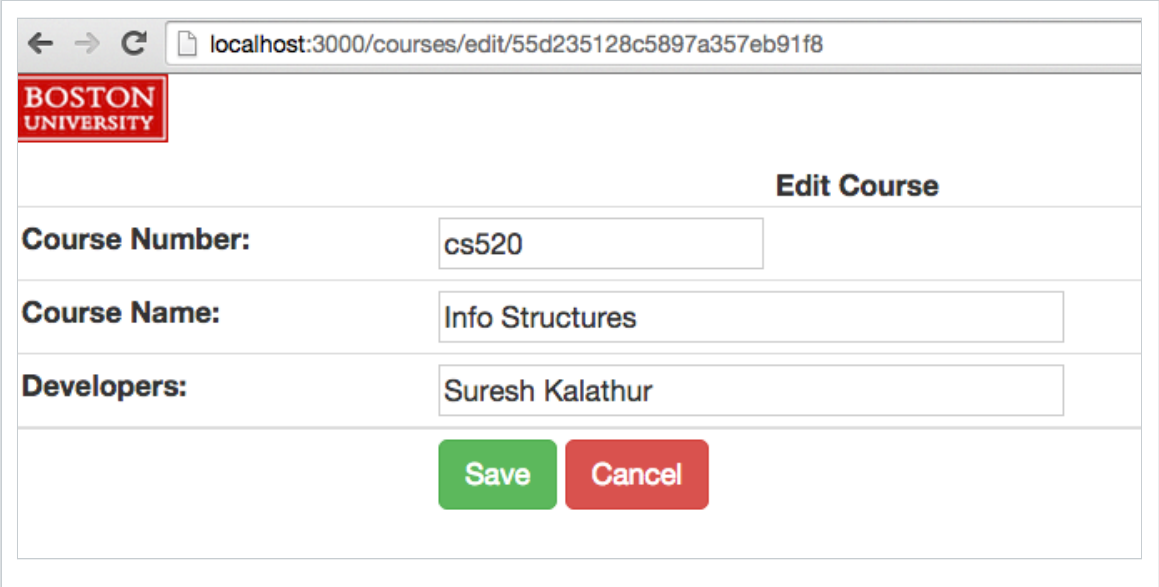
```
1  var DB = require('./dbConnection.js');
2  var Course = DB.getModel();
3
4  module.exports =
5    function editCourse(req , res , next){
6      var id = req.params.id;
7
8      Course.findById(id, function (err, course){
9        if(err)
10         console.log("Error Selecting : %s ", err);
11       if (!course)
12         return res.render('404');
13       res.render('editCourseView',
14           {title:"Edit Course",
15            data: {id: course._id,
16                   courseNumber: course.courseNumber,
17                   courseName: course.courseName,
18                   developers:  course.getDeveloperNames()}
19           });
20     });
21 };
```

The view page for editing the selected course is shown below. In addition to the inputs for the course name, course number and course developer, buttons are provided for saving the course data or cancel the operation.

editCourseView.handlebars

```
<div class="table-responsive">
  <form method="post" action='/courses/edit/{{data.id}}'>
    <table class="table table-condensed">
      <tbody>
        <tr>
          <th width="200">Course Number: </td>
          <td><input type="text" name="cnumber" required
                value="{{data.courseNumber}}"></td>
        </tr>
        <tr>
          <th>Course Name: </td>
          <td><input type="text" name="cname" required size="40"
                value="{{data.courseName}}"></td>
        </tr>
        <tr>
          <th>Developers: </td>
          <td><input type="text" name="cdev" size="40"
                value="{{data.developers}}"></td>
        </tr>
      </tbody>
      <tfooter>
        <tr>
          <td></td>
          <td>
              <input type="submit" value="Save" class="btn btn-success">
              <input type="button" value="Cancel" class="btn btn-danger"
                class="cancel" onclick="cancelAdd()">
          </td>
        </tr>
      </tfooter>
    </table>
  </form>
</div>
```

The layout for the web page while editing the course is shown below. The existing values are shown in the input fields.

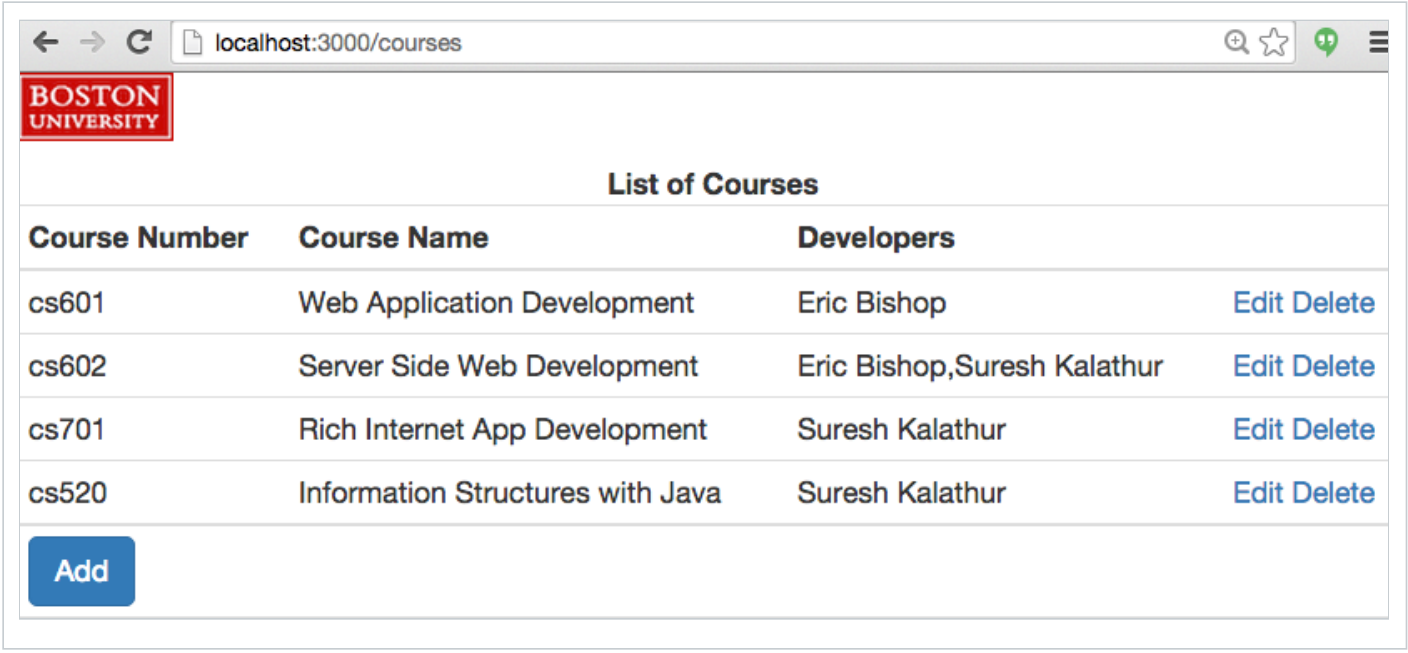


When the user submits this form for the modifications, the *saveAfterEdit* module exports the required functionality as shown below. The course number, course name and course developers are updated for the specified *id*. The former values are obtained through the request's body while the latter is retrieved from the request's parameters.

saveAfterEdit.js

```
var DB = require('./dbConnection.js');
var Course = DB.getModel();

module.exports =
  function saveCourse(req , res , next){
    var id = req.params.id;

    Course.findById(id, function (err, course){
      if(err)
        console.log("Error Selecting : %s ", err);
      if (!course)
        return res.render('404');

        course.courseNumber = req.body.cnumber
        course.courseName = req.body.cname;
        var developers = req.body.cdev;
        if (developers.length > 0) {
          developers =
            developers.split(',').map(function (elem){
              var names = elem.trim().split(' ');
              return {firstName: names[0],
                      lastName: names[1]};
            });
          course.courseDevelopers = developers;
        }

        course.save(function (err) {
          if (err)
            console.log("Error updating : %s ",err );
          res.redirect('/courses');
        });
    });
  };
```

After the update is successful, the application redirects the user to the course list as shown below.

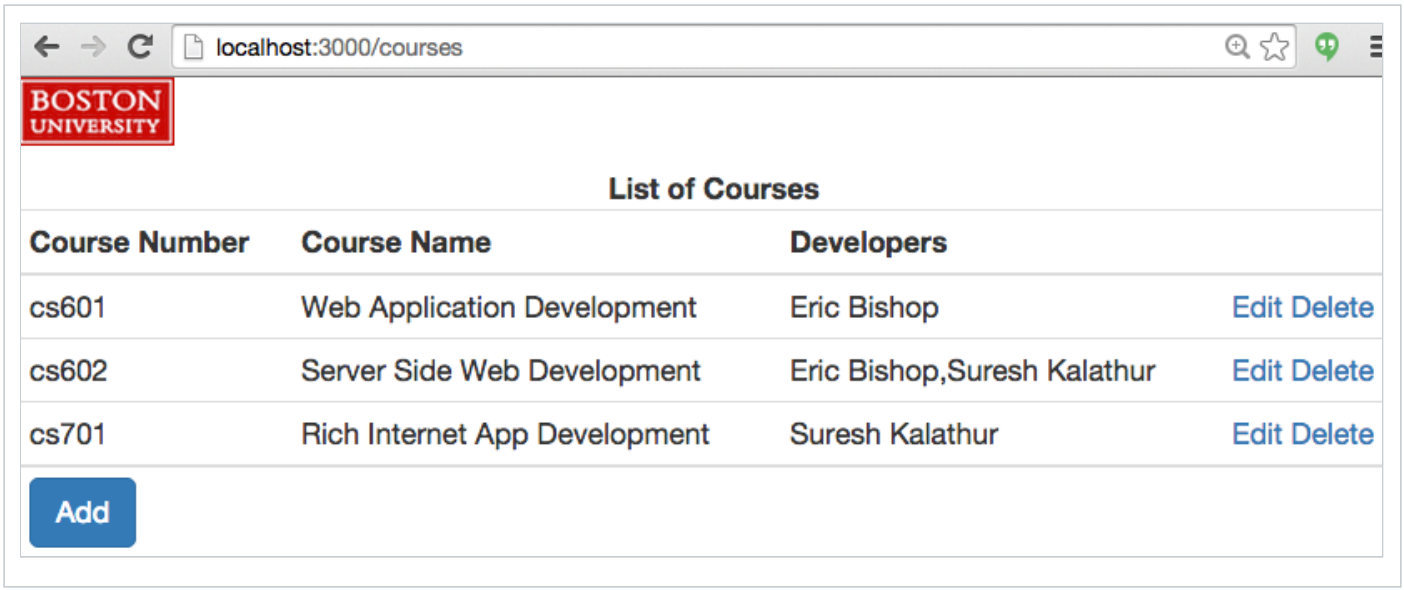


The *deleteCourse* module exports the function that deletes the specified course whose *id* is specified in the request's parameters.

deleteCourse.js

```
1  var DB = require('./dbConnection.js');
2  var Course = DB.getModel();
3
4  module.exports =
5    function deleteCourse(req , res , next){
6      var id = req.params.id;
7
8      Course.findById(id, function (err, course){
9        if(err)
10         console.log("Error Selecting : %s ", err);
11       if (!course)
12         return res.render('404');
13
14       course.remove(function (err) {
15         if (err)
16           console.log("Error deleting : %s ",err );
17         res.redirect('/courses');
18       });
19     });
20   };
```

After the delete is successful, the application redirects the user to the course list as shown below.



Bibliography

MongoDB Data Models, https://docs.mongodb.org/manual/data-modeling/

MongoDB Node.js Driver, https://github.com/mongodb/node-mongodb-native

Mongoose: elegant object mongodb modeling for node.js, http://mongoosejs.com

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