

WA2452 Node.js Software Development

Student Labs

Web Age Solutions Inc.

Table of Contents

Lab 1 - Getting Started With Node.js.....	3
Lab 2 - Basics of a Node.js Module.....	5
Lab 3 - Using Node Package Manager (npm).....	8
Lab 4 - Building Module Dependency.....	12
Lab 5 - Using the Stream API.....	16
Lab 6 - Events in Node.js.....	19
Lab 7 - Asynchronous Programming with Callbacks.....	26
Lab 8 - Asynchronous Programming with Promises.....	33
Lab 9 - Basic Web Application Development.....	42
Lab 10 - Debugging a Node.js Application.....	47
Lab 11 - Introduction to Unit Testing.....	51
Lab 12 - Logging with Morgan.....	56
Lab 13 - Web Service Using Express.....	63
Lab 14 - Using MongoDB.....	68
Lab 15 - Using the Jade Template Engine.....	78
Lab 16 - Clustering a Node.js Application.....	84
Lab 17 - MicroServices with Node.....	89
Lab 18 - Test RESTful API with Supertest.....	109
Lab 19 - Mock RESTful API with Nock.....	115

Lab 1 - Getting Started With Node.js

In this lab we will explore how Node.js is installed and create a very simple Node.js application.

Part 1 - Explore the Installation

To save time Node.js is already installed for you in your desktop. Let's verify to make sure things are installed correctly.

- __1. Open a command prompt window.
- __2. Enter this command to check the version of Node.js installed.

```
node --version
```

- __3. In Windows installing Node.js also installs npm the package manager. Enter this command to check its version.

```
npm --version
```

- __4. Open file explorer. Go to **C:\Program Files\nodejs** folder. This is where Node.js is installed. This folder is added to the PATH environment variable by the installer so that you can run the **node** and **npm** commands from anywhere.

Part 2 - Create a Command Line Application

We will now create a very simple command line application using Node.js.

- __1. In the command prompt window change directory to **C:\LabFiles**.

```
cd C:\LabFiles
```

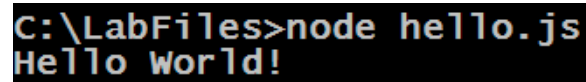
- __2. In that folder create a file called **hello.js**. Visual Studio Code has been provided for you to use when editing code and related files. This editor includes excellent support for JavaScript files. That being said, feel free to use any editor that you are comfortable with.
- __3. In hello.js enter this line.

```
console.log("Hello World!");
```

- __4. Save the file.

__5. From the command prompt enter this command to execute the program.

```
node hello.js
```



```
C:\LabFiles>node hello.js
Hello world!
```

It's that simple!

Let's make things a little more interesting. How about we supply the name of the planet to say hello as a command line argument? In Node.js you can access the command line parameters using the **process** global variable.

__6. Change the code of your program as follows.

```
var planet = "World"; //Default planet

if (process.argv.length > 2) {
  planet = process.argv[2]; //3rd parameter
}

console.log("Hello %s!", planet);
```

__7. Save changes.

__8. Try running the program in these ways.

```
node hello.js
```

```
node hello.js Mars
```

Even though JavaScript is an interpreted language, the V8 JavaScript engine used by Node.js has a Just in Time compiler that can compile certain sections of your code for better performance.

__9. Close the editor.

__10. Leave the command prompt open.

Part 3 - Review

In this short lab we reviewed the installation of Node.js. We also wrote a simple application and executed it.

Lab 2 - Basics of a Node.js Module

Modules let you break up a large application in multiple files. You can also download third party modules and use them. You can create file based or directory based modules. We will learn to do them both.

Part 1 - Create a File Based Module

We will create a module that encapsulates the authentication logic of our application. This module will be used by our application to log users in.

__ 1. Within the **C:\LabFiles** folder create a new file called **auth.js**.

__ 2. Enter this code:

```
module.exports.login = function(userId, password) {  
  //Awesome auth logic  
  if (userId === "bob" && password === "pass123") {  
    return true;  
  }  
  
  return false;  
}
```

Any function or variable that is exposed to the user of a module is added as a property of **module.exports**. We did just that.

__ 3. Save and close the file.

Part 2 - Use the Module

We will now use the auth module from another file.

__ 1. In the **C:\LabFiles** folder create a new file called **awesome_app.js**.

__ 2. In that file enter this line to load the auth module.

```
var auth = require("../auth");
```

The `require()` function takes a path to the module file minus the `.js` extension.

__ 3. Add these lines to the file.

```
function test_login(userId, password) {  
  if (auth.login(userId, password)) {  
    console.log("%s logged in fine!", userId);  
  } else {  
    console.log("%s can't login.", userId);  
  }  
}  
  
test_login("daffy", "d12");  
test_login("bob", "pass123");
```

__ 4. Save changes.

Part 3 - Test

__ 1. At the command prompt enter this command.

```
node awesome_app.js
```

__ 2. Make sure that you see this output.

```
daffy can't login.  
bob logged in fine!
```

Part 4 - Make a Directory Based Module

A large scale module is better created as a directory based module. This lets you break up that large module into submodule files.

The name of the directory makes up the module's name. It needs to have a file called **index.js** that serves as the entry point. When you load this module it is the index.js file that gets loaded first.

We will now convert the auth module into a directory based module.

__ 1. In the command prompt enter this command to create the **C:\LabFiles\auth** folder.

```
mkdir auth
```

__2. Rename auth.js to index.js.

```
rename auth.js index.js
```

__3. Move index.js inside the **auth** folder.

```
move index.js auth
```

That's it. We don't have to change awesome_app.js. The call to require() function will still work. This time it will load the module by the directory name.

Part 5 - Test

__1. At the command prompt enter this command.

```
node awesome_app.js
```

__2. Make sure that you see this output.

```
daffy can't login.  
bob logged in fine!
```

__3. Close the editor.

__4. Leave the command prompt open.

Part 6 - Review

In this lab we learned how to create a file and directory based module and use that module from another module. The key is to add any public function or variable as properties of the **module.exports** object.

Lab 3 - Using Node Package Manager (npm)

npm has several key responsibilities:

- 1) Download and install modules from a central repository.
- 2) Manage dependency between modules.

Dependencies are transitive. That means if module A depends on B and B depends on C then installing B will automatically install module C. Module A does not have to declare its eventual dependency on C.

In this lab we will install the **uuid** module. This is used to generate a universally unique ID (UUID) as per RFC4122.

Part 1 - Can you Use npm?

If access to the Internet is behind a proxy server then configure proxy as follows:

```
npm config set proxy http://"user:password"@proxy.company.com:1234  
npm config set https-proxy http://"user:password"@proxy.company.com:1234
```

If npm still fails in these labs there is a fallback. You will see a file called **C:\LabFiles\all_packages.zip**. You will need to unzip it in the folder of the module where you are working on. This will give you all the modules that you need to complete the labs.

Part 2 - Create Application Directory

We will now create a directory for our work.

- __1. Inside C:\LabFiles create a folder called **unique**.

Part 3 - Install the uuid Package

- __1. In the command prompt window change directory to unique.

```
cd C:\LabFiles\unique
```

- __2. Run this command to install the chalk package.

```
npm install uuid@2.0.1
```


You should see an output like this.

```
C:\LabFiles\unique>npm install uuid@2.0.1
uuid@2.0.1 ..\node_modules\uuid
```

If npm is failing to install the package because of Internet connection problems, extract C:\LabFiles\all_packages.zip inside of C:\LabFiles\unique to obtain the packages.

__3. Verify that a folder called **node_modules** is created within **C:\LabFiles\unique**. This is where packages that are local to the unique module will be stored.

Troubleshooting. If there was already a folder called *C:\LabFiles\node_modules* before running the command then the install will be installed in *C:\LabFiles\node_modules* instead on *C:\LabFiles\unique\node_modules*. If so then rename the *C:\LabFiles\node_modules* to *C:\LabFiles\node_modulesA*, run the command again and verify *C:\LabFiles\unique\node_modules* was created.

__4. Run this command to list all installed local packages.

```
npm ls --depth=0
```

You should see that the uuid package installed.

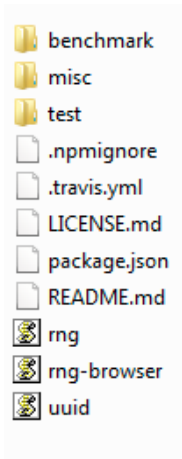
Part 4 - Inspect the Package

So what does an npm package look like? And what is the difference between an npm package and a Node.js module?

A package is any directory that has the **package.json** file. This means a Node.js directory based module that has package.json will qualify as an npm package. You can also have packages that have nothing to do with Node.js. For example you can get browser side JavaScript libraries distributed as npm packages.

__1. Using file browser inspect the contents of the **C:\LabFiles\unique\node_modules** folder. Verify that you see the **uuid** subfolder.

__2. Navigate inside the **uuid** folder.



We see the package.json file as we expected. But where is the index.js file?

__3. Open **package.json** in an editor.

__4. Locate this line:

```
"main": "./uuid.js",
```

This tells Node.js that the main entry point file of this module is uuid.js (and not the default index.js).

Part 5 - Use the Package

__1. Inside **C:\LabFiles\unique** create a file called **index.js**.

__2. In that file add these lines.

```
var uuid = require("uuid");

var value = uuid.v4(); //Generate v4 uuid
console.log("UUID is: %s", value);
```

Note that to load packages installed by npm we use the name of the module and not its path. When you do not use path (that is the string supplied to require()) doesn't start with / or ./, node will look for the module in the node_modules folder. That is also where npm stores the packages.

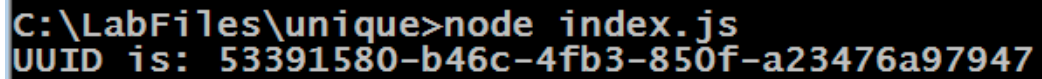
__3. Save changes.

Part 6 - Test

__1. From the command prompt run:

```
node index.js
```

__2. Verify that the program works. Your UUID will be different.



```
C:\LabFiles\unique>node index.js
UUID is: 53391580-b46c-4fb3-850f-a23476a97947
```

__3. Close all open files in the editor.

__4. Leave the command prompt open.

Part 7 - Review

In this lab we learned how to use npm to install packages from a central repository. We also learned how to load and use such packages.

Lab 4 - Building Module Dependency

In the previous lab we used the uuid module from our unique module. But we manually installed the uuid module. This works fine but when your module depends only a few external modules. If you have dependency on many modules manually installing modules become tedious and error prone. Everyone in the development team will have to carefully install all the modules every time they clone the code repository. A better option will be to formally specify the dependency in the package.json file. We will now learn to do that.

Part 1 - Uninstall the uuid Package

__1. In the command prompt window change directory **C:\LabFiles\unique** if you are not already there.

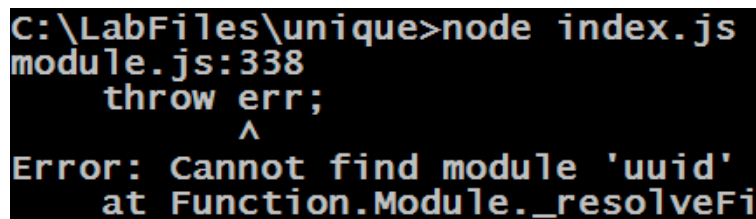
__2. Run this command to uninstall the uuid package.

```
npm uninstall uuid
```

__3. Try to run the program again.

```
node index.js
```

It should fail.



```
C:\LabFiles\unique>node index.js
module.js:338
    throw err;
    ^
Error: Cannot find module 'uuid'
at Function.Module._resolveFi
```

Part 2 - Specify the Dependencies

Dependency is stated in the package.json file. We can manually create the file. Only the name and version fields are mandatory. But it is generally easier to generate it using npm.

__1. While still in the C:\LabFiles\unique folder enter this command.

```
npm init
```

__2. The name should default to unique. Just hit enter to continue.

__3. Accept 1.0.0 as the version. Hit enter to continue.

___ 4. For all remaining fields, simply hit enter to continue.

```
{  
  "name": "unique",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"  
  },  
  "author": "",  
  "license": "ISC"  
}  
  
Is this ok? (yes)
```

___ 5. Hit enter to accept the changes. This will create the package.json file.

___ 6. Open the file in an editor.

___ 7. Add the dependency as shown in bold face below.

```
"main": "index.js",  
"dependencies": {  
  "uuid": "2.0.1"  
},  
"scripts": {
```

___ 8. Save changes and close the file.

___ 9. From command line enter this command to install the required packages.

```
npm install
```

```
C:\LabFiles\unique>npm install  
npm WARN package.json unique@1.0.0 No description  
npm WARN package.json unique@1.0.0 No repository field  
npm WARN package.json unique@1.0.0 No README data  
uuid@2.0.1 node_modules\uuid
```

___ 10. You will get a few warnings. They can be ignored.

Part 3 - Test

__1. Try to run the program again.

```
node index.js
```

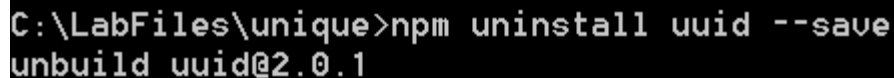
__2. Make sure that it works this time.

Part 4 - Use npm to Maintain Dependencies

So far we have used "*" as the required version for uuid. This means that the latest major version will do. In real life grabbing whatever is the latest version may break your code. You may want to pin down the version number. This process is simplified by using npm to manage the dependencies.

__1. Run this command to uninstall uuid as well as remove it from package.json.

```
npm uninstall uuid --save
```



```
C:\LabFiles\unique>npm uninstall uuid --save
unbuild uuid@2.0.1
```

__2. Verify that the module has been removed from package.json and close the file.

```
"dependencies": {},
```

__3. Now run this command to install uuid again and automatically add it to package.json.

```
npm install uuid@2.0.1 --save
```

__4. The dependencies section in package.json will now look like this. Review and close the file.

```
"dependencies": {
  "uuid": "^2.0.1"
},
```

The use of ^ will pin down the major version of the module.

Generally speaking use npm to maintain the dependencies section of package.json.
--

Part 5 - Test

__1. Try to run the program again.

```
node index.js
```

__2. Make sure that it works this time.

__3. Close the editor.

__4. Leave the command prompt open.

Part 6 - Review

In this lab we learned how to specify all dependencies of a module in its package.json file. We also learned how to use npm to maintain the dependencies section of that file.

Lab 5 - Using the Stream API

The Stream API provides a generic way to represent any readable and writable device. Such a device can be a file on disk or a network socket. There are many benefits of using the Stream API.

- 1) Provides an efficient way to read and write data. Instead of reading the entire data from a device in memory you will read small amounts of data as it becomes available.
- 2) Since many Node.js modules consistently use this API all kinds of interesting possibilities appear. For example with only a few lines of code you can write an entire file to a network socket in a very memory efficient manner.

In this lab we will use the Stream API to read files line by line. After each line is read we will invoke a callback function and supply that line as a parameter. We will develop this code in a module and then use the module from elsewhere.

Part 1 - Create the readline Module

- ___ 1. Create a folder called **liner** in **C:\LabFiles**.
- ___ 2. In the liner folder create a file called **readline.js**.
- ___ 3. In that file add this line to import the **fs** (File system) module.

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

- ___ 4. Then add these lines to create the shell of the method that will read a file line by line.

```
module.exports.readLine = function (fileName, callback) {  
  }  
}
```

- ___ 5. Within that method add this line to create a new Reader object from the file.

```
var input = fs.createReadStream(fileName);
```

Now we will write the code to read from the file. In StreamAPI the Reader object fires an event called "data" when any data becomes available to read in the device. When the device is closed and no more data is available then the "end" event is fired. We will take advantage of these events to do our reading.

___ 6. Within the method add these lines to complete implementation.

```
var textLeft = '';

input.on('data', function(data) {
    textLeft += data;

    var index = textLeft.indexOf('\n');

    while (index > -1) {
        var line = textLeft.substring(0, index);
        textLeft = textLeft.substring(index + 1);
        callback(line);
        index = textLeft.indexOf('\n');
    }
});

input.on('end', function() {
    if (textLeft.length > 0) {
        callback(textLeft);
    }
});
```

We will leave you to understand the actual algorithm used to process the data and cut it into lines. The important thing to note here is that we never read the whole file into memory. As a result this code will work fairly efficiently.

The **data** variable passed to the "data" event handler function is a Buffer object. But the **textLeft** variable is a string. When we add data to textLeft using the + operator the Buffer gets converted to a string. This will work only if the Buffer contains valid UTF-8 data.

___ 7. Save changes.

Part 2 - Use the Module

___ 1. Create a file called **index.js** in the **liner** folder.

___ 2. Add these lines to that file.

```
var rl = require("./readline");

rl.readLine(process.argv[2], function(line) {
    console.log("Read line: %s", line);
});
```

We intend to supply the name of the file in command line.

___ 3. Save changes.

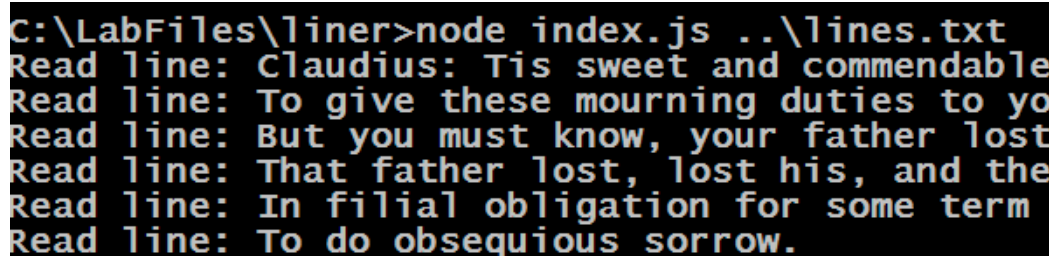
Part 3 - Test

__1. In the command prompt window switch directory to C:\LabFiles\liner.

```
cd C:\LabFiles\liner
```

__2. Enter this command to run your program.

```
node index.js ../lines.txt
```



```
C:\LabFiles\liner>node index.js ../lines.txt
Read line: Claudius: Tis sweet and commendable
Read line: To give these mourning duties to yo
Read line: But you must know, your father lost
Read line: That father lost, lost his, and the
Read line: In filial obligation for some term
Read line: To do obsequious sorrow.
```

__3. Verify that the program output looks like above.

__4. Close the editor.

__5. Leave the command prompt open.

Part 4 - Review

In this lab we got a taste of the Stream API. This API is used heavily from many Node.js modules. Here we used the API to read from files. A couple of best practices used in this lab are:

- 1) Create reusable code in its own module.
- 2) Use callbacks to asynchronously read lines from files.
- 3) Minimize the use of memory when reading by using the Stream API.

Lab 6 - Events in Node.js

In this lab you will utilize event emitter and event handler in Node.js.

The lab takes a simple banking scenario. BankAccount class will be defined in bank_account.js module. This class will contain a variable for storing the account balance. The class will also contain withdraw function which will deduct "amount" from the account balance. If the resulting balance is a negative value, event emitter will be used to emit a custom "low_balance" event. This emitted event will be handled in main.js module.

Part 1 - Create Directory Structure

In this part you will create directory structure for storing event emitter application

__ 1. Open **Command Prompt** from the **Start Menu**.

__ 2. Switch to the **Workspace** directory

```
cd c:\workspace
```

Note: If the directory doesn't exist, create it by using md [c:\workspace](#) and then switch to it by executing the above command.

__ 3. Create a directory named "events" by using following command

```
md events
```

__ 4. Switch to the newly created directory

```
cd events
```

Part 2 - Create bank_account.js Module

In this part you will create bank_account.js module. It will import **events** module to use Event Emitter to emit "low_balance" event when the balance goes below 0.

__ 1. Use a text editor to create a blank file named **bank_account.js** under **events** directory.

__ 2. In the bank_account.js file, type following code

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

Note: This code imports "events" module. It contains Event Emitter related functions which you will utilize later in the lab.

__3. Below the above line, add following code to create a **BankAccount** function (ES5 equivalent of class).

```
var BankAccount = function BankAccount() {  
  
}
```

Note: It's currently an empty function. You will add variables and functions to it later in this part of the lab.

__4. Below the above function (end of file), add following code

```
module.exports.BankAccount = BankAccount;
```

Note: This code exports the above created function (ES5 equivalent of class), so it can be reused by Node.js application.

__5. Inside the BankAccount function, add following code

```
var balance = 0;
```

Note: This line declares a variable which will store the account balance.

__6. Below the above variable, add following code

```
this.eventEmitter = new events.EventEmitter();
```

Note: This line utilizes the events module to create an instance of Event Emitter. Notice you are using the **this** keyword to create eventEmitter as a public property. It will be accessed, later in the lab, by main.js module to create an event handler.

__7. Below to above Event Emitter code, add following method

```
this.deposit = function(amount) {  
    balance += amount;  
}
```

Note: This method deposits a certain amount to the account balance.

__8. Below the deposit function, add following code

```
this.withdraw = function(amount) {  
    var newBalance = balance - amount;  
    if(newBalance < 0) {
```

```

        this.eventEmitter.emit("low_balance", balance, newBalance);
    } else {
        balance -= amount;
    }
}

```

Note: This code manages the withdraw operation. It checks whether the new balance will become a negative value. If yes, then it emits (fires or triggers) the "low_balance" event, otherwise, it deducts the amount from the balance.

__9. Save the file and close the text editor.

Part 3 - Create main.js Module

In this part you will create main.js module which will utilize the bank_account.js module. The main.js module will instantiate the BankAccount class, deposit 100 to the account balance then withdraw 110 from the balance. It will also create an event handler for the "low_balance" event, which will get triggered when the resulting balance becomes a negative value.

__1. Using a text editor, create a file named main.js under **events** directory.

__2. Add following code to the main.js module

```
var bank_account = require('./bank_account.js');
```

Note: This code imports bank_account.js module.

__3. Below the require statement, add following code

```
var account = new bank_account.BankAccount();
```

Note: This code instantiates the BankAccount object and stores it in **account**.

__4. Below the above line, add following code

```
account.eventEmitter.on("low_balance", function(currentBalance,
newBalance) {
    console.log('Low balance!!!. Current balance is: ' + currentBalance +
", Balance will become: " + newBalance);
});
```

Note: The above code subscribes (creates an event handler) to the "low_balance" event. It receives the parameters from the event emitter and displays them on the console.

__5. Below the above event handling code, add following code

```
account.deposit(100);  
account.withdraw(110);
```

Note: The above code deposits 100 to the balance and then withdraws 110 from it. Withdraw operation should result in emission of the event since the resulting balance will become a negative value.

__6. Save the file and close the text editor.

Part 4 - Test the code

In this part you will test the modules you created in the previous parts of this lab.

__1. In **Command Prompt** window, under **events** directory, run following command

```
node main.js
```

Notice it displays the message "Low balance!!!. Current balance is: 100, Balance will become: -10"

Part 5 - Creating Multiple Event Handlers (Listeners)

In this part you will add another event handler by duplicating the existing one. You will test the code then limit the maximum event handlers to 1 and retest the code.

__1. Open the file 'main.js' in a text editor.

__2. Locate the following code:

```
account.eventEmitter.on(...)
```

__3. Duplicate the function by copying the existing one and change console.log message as shown in the bold text below

```
account.eventEmitter.on("low_balance", function(currentBalance,  
balanceWillBe) {  
  console.log('MANAGER - Low balance!!!. Current balance is: ' +  
currentBalance + ", Balance will become: " + balanceWillBe);  
});  
  
account.eventEmitter.on("low_balance", function(currentBalance,  
balanceWillBe) {  
  console.log('CUSTOMER - Low balance!!!. Current balance is: ' +  
currentBalance + ", Balance will become: " + balanceWillBe);  
});
```

```
});
```

Note: You have two event handlers / listeners now. One is for the bank manager and second is for the customer.

__ 4. Save the file and close the text editor.

__ 5. In **Command Prompt** window, run **main.js** file

```
node main.js
```

Notice the result looks like this

```
MANAGER - Low balance!!!. Current balance is: 100, Balance will become: -10  
CUSTOMER - Low balance!!!. Current balance is: 100, Balance will become: -10
```

Both event handlers got called when "low_balance" custom event got emitted / fired.

__ 6. Open 'bank_account.js' in a text editor.

__ 7. Locate the following statement:

```
this.eventEmitter = new events.EventEmitter();
```

__ 8. Below the above statement, add following code

```
this.eventEmitter.setMaxListeners(1);
```

Note: You are limiting the maximum event handlers / listeners to 1.

__ 9. Save the file.

__ 10. In **Command Prompt** window, run **main.js** file

```
node main.js
```

Notice It shows a warning that there's a potential memory leak since more than 1 listeners are detected. After the warning, it shows messages for both the event handlers. The result looks like this.

```
(node) warning: possible EventEmitter memory leak detected. 2 low_balance listeners added. Use emitter.setMaxListeners()
to increase limit.
Trace
  at EventEmitter.addListener (events.js:239:17)
  at Object.<anonymous> (C:\workspace\events\main2.js:11:22)
  at Module._compile (module.js:409:26)
  at Object.Module._extensions..js (module.js:416:10)
  at Module.load (module.js:343:32)
  at Function.Module._load (module.js:300:12)
  at Function.Module.runMain (module.js:441:10)
  at startup (node.js:140:18)
  at node.js:1043:3
MANAGER - Low balance!!!. Current balance is: 100, Balance will become: -10
CUSTOMER - Low balance!!!. Current balance is: 100, Balance will become: -10
```

__ 11. Close the text editor and **Command Prompt** window(s)

Part 6 - Using a Better Pattern for using EventEmitter

Previously, you used EventEmitter as an object inside your custom function / class. Although it worked, but better practice is to inherit from the EventEmitter class and then utilize its functionalities. In this part, you will follow the best practice for using EventEmitter.

- __ 1. In a text editor, open `bank_account.js` file
- __ 2. Below the existing `require` statement, add following code

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

Note: `util` module will be used for easily inheriting from an existing class.

- __ 3. Below the existing `var BankAccount = function BankAccount() {...}` code, add following code

```
util.inherits(BankAccount, events.EventEmitter);
```

Note: This statement inherits `BankAccount` from the `EventEmitter` class.

- __ 4. Inside the `BankAccount` function, remove the following statement

```
this.eventEmitter = new events.EventEmitter();
```

- __ 5. Inside the `BankAccount` function, change `this.eventEmitter.setMaxListeners(1);` to following

```
this.setMaxListeners(1);
```

- __ 6. Inside the `BankAccount` function, change `this.eventEmitter.emit(...);` to following


```
this.emit("low_balance", balance, newBalance);
```

__7. Save the file and close the text editor.

__8. In a text editor, open main.js file

__9. Below the existing require statements, add following code

```
var events = require('events');  
var util = require('util');
```

__10. Below **var account = ...** statement, add following code

```
util.inherits(account, events.EventEmitter);
```

Note: This statement inherits account object from EventEmitter.

__11. Find **account.eventEmitter.on(...)**; function calls and remove the word eventEmitter so they look like this:

```
account.on(...);
```

__12. Save the file and close the text editor.

__13. In Command Prompt window, run main.js file

```
node main.js
```

Notice the result is same as the previous part of this lab.

Part 7 - Review

In this lab you utilized event emitter and event handling feature of Node.js.

Lab 7 - Asynchronous Programming with Callbacks

In this lab you will utilize callback functions in synchronous and asynchronous ways. You will also see how to create a custom asynchronous callback function.

Part 1 - Create Directory Structure

In this part you will create directory structure for storing callbacks application

__ 1. Open **Command Prompt** from the **Start Menu**.

__ 2. Switch to the **Workspace** directory

```
cd c:\workspace
```

Note: If the directory doesn't exist, create it by using md [c:\workspace](#) and then switch to it by executing the above command.

__ 3. Create a directory named "callbacks" by using following command

```
md callbacks
```

__ 4. Switch to the newly created directory

```
cd callbacks
```

__ 5. Using a text editor, create a file named **input.txt** under the **callbacks** directory with following contents

```
Line 1  
Line 2  
Line 3
```

__ 6. Save the file and close the text editor

Part 2 - Create util.js Module

In this part you will create util.js module. It will define **pause** function and export it. This module will be used by synchronous and asynchronous callback functions, later in the lab.

__ 1. Use a text editor to create a blank file named **util.js** under **callbacks** directory.

__ 2. In the util.js file, type following code

```
function pause(ms) {  
    var waitTill = new Date(new Date().getTime() + ms);  
    while(waitTill > new Date()) {}  
}
```

Note: The pause function uses a while loop to block code for **ms** milliseconds.

__3. Below the above code, add following code

```
module.exports.pause = pause;
```

Note: This line exports the pause function.

__4. Save the file and close the text editor.

Part 3 - Utilize Synchronous Programming Technique to Read File Contents

In this part you will utilize synchronous programming technique, i.e. blocking code, to read file contents and display them on the screen. You will also utilize the **pause** function from **util.js** module, which you created in the previous part of the lab.

__1. Using a text editor, create a new file named **file_sync.js** under **callbacks** directory.

__2. Add following code to the file

```
var fs = require("fs");  
var util = require("../util");
```

Note: These lines import the built-in fs and the custom util modules.

__3. Below the above code, add following code

```
try {  
  
} catch (err) {  
    console.log(err);  
}
```

Note: The above lines define a try-catch block and display error message on the screen.

__4. In the try block, add following code

```
var data = fs.readFileSync('input.txt');
```

```
console.log("Waiting for 3 seconds...");
util.pause(3000);

console.log(data.toString());
```

The code uses synchronous function for reading file contents, pauses for 3 seconds, then displays the file contents on the screen.

__5. After the catch block, the end of file, add following code

```
console.log("This line will get called AFTER file contents are
displayed!");
```

Note: This line displays a message on the screen.

__6. Save the file and close the text editor

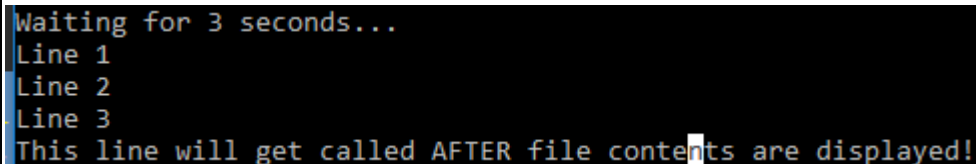
__7. In **Command Prompt**, switch to the **callbacks** directory

```
cd c:\workspace\callbacks
```

__8. Run 'file_sync.js' using Node:

```
node file_sync.js
```

Notice the result looks like this



```
Waiting for 3 seconds...
Line 1
Line 2
Line 3
This line will get called AFTER file contents are displayed!
```

The code pauses for 3 seconds, displays the file contents, then displays a message on the screen. It's the expected output since you have used synchronous programming technique. In the next part of this lab, you will use asynchronous callback technique which will change the display order.

Part 4 - Utilize Asynchronous Programming Technique to Read File Contents

In this part you will rewrite the above code to utilize asynchronous callbacks to read file contents. It will change the display order of the final output.

- __1. Using a text editor, create a new file named **file_async.js** under **callbacks** directory
- __2. Add following code to the file

```
var fs = require("fs");  
var util = require("./util");
```

Note: These lines import the built-in fs and the custom util modules

- __3. Below the above lines, add following code

```
fs.readFile('input.txt', function (err, data) {  
  if (err)  
    return console.error(err);  
  console.log("Waiting for 3 seconds...");  
  util.pause(3000);  
  
  console.log(data.toString());  
});
```

Note: It's a single asynchronous readFile function call. It reads input.txt file and executes a callback function when the file contents are read. The callback function receives two arguments: error / exception, if there's any, and the file contents in data argument. The callback function pauses for 3 seconds then displays the file contents.

- __4. Below the readFile function call (end of file), add following code

```
console.log("This line will get called BEFORE file contents are  
displayed!");
```

Note: The above line displays a message on the screen.

- __5. Save the file and close the text editor
- __6. In the **Command Prompt** window, switch to the **callbacks** directory, if you haven't already done so.

```
cd c:\workspace\callbacks
```

- __7. Run the file_async.js file

```
node file_async.js
```

Notice the output looks like this:

```
This line will get called BEFORE file contents are displayed!
Waiting for 3 seconds...
line 1
line 2
line 3
```

Notice even though `console.log` statement is written after the `readFile` function call, it still gets called before the `readFile` function. The reason is `readFile` is using asynchronous callback technique which doesn't block the code. The `pause` function blocks the code since you have used it inside the callback.

Part 5 - Create a Custom Asynchronous Callback Function

In this part you will create a function named **sum** which will utilize asynchronous / non-blocking callback function

- __ 1. Using a text editor, create a new file named **sum.js** under the **callbacks** directory
- __ 2. Add following code to the file

```
function sum(a, b, callback) {
    var err = null;
    if(a < 0 || b < 0)
        err = "Argument must not be negative!";
    var result = a + b;
    callback(err, result);
}
```

Note: The function takes 3 arguments: the numbers to add up and a callback function which will be called when result is ready. The function also handles error by checking the argument values. If one of the arguments is a negative value, it sets the error message. Currently, the callback function won't get called asynchronously. It's still a synchronous callback function. You will add more code to it later in the lab.

- __ 3. Below the above function definition, add the following code

```
module.exports.sum = sum;
```

Note: This code exports the `sum` function

- __ 4. Save the file and close the text editor.
- __ 5. Using the text editor, create a new file named **main.js** under the **callbacks** directory
- __ 6. Add following code to the file

```
var util = require('./util');
```

```
var sum = require('./sum');
```

Note: These lines import custom modules named util and sum.

__7. Below the above code, add the following code

```
sum.sum(5,6, function(err, data) {  
    if(err) {  
        console.log(err);  
    } else {  
        console.log("Pausing for 3 seconds...");  
        util.pause(3000);  
  
        console.log("The result is: " + data);  
    }  
});
```

Note: Here you are calling the custom sum function defined in the sum module. You are passing it 5 and 6 as arguments. You are also passing it a callback function which will get called when the result is returned by the sum function. The callback function receives the result in data variable. It pauses for 3 seconds and displays the result on the console.

__8. Below the above code, add the following code

```
console.log("This line will get called BEFORE sum is called!");
```

__9. Save the file and close the text editor

__10. In **Command Prompt** window, navigate to the **callbacks** directory

```
cd c:\workspace\callbacks
```

__11. Run main.js

```
node main.js
```

Notice the result is looks like this

```
Pausing for 3 seconds...  
The result is: 11  
This line will get called BEFORE sumAsync is called!
```

__12. The result is synchronous right now. It paused for 3 seconds, displayed the result,

then displayed a message on the console.

__13. Open sum.js file in a text editor

__14. Modify the **sum** function definition so it looks like this (Note: Changes are in bold)

```
function sum(a, b, callback) {  
    var err = null;  
    if(a < 0 || b < 0)  
        err = "Argument must not be negative!";  
  
    var result = a + b;  
  
    setImmediate(function () {  
        callback(err, result);  
    });  
}
```

__15. Save the file and close the text editor.

__16. Run main.js file again

```
node main.js
```

Notice the result looks like this

```
This line will get called BEFORE sumAsync is called!  
Pausing for 3 seconds...  
The result is: 11
```

This time the callback function got called asynchronously. It displays the message first, paused for 3 seconds, then displayed the result on the screen.

Part 6 - Review

In this lab you utilized callback functions in synchronous and asynchronous ways. You also saw how to create a custom asynchronous callback function.

Lab 8 - Asynchronous Programming with Promises

In this lab you will utilize **promises** to make asynchronous function calls without relying on callback function. You will also use Bluebird module to promisify existing callback functions to make them asynchronous.

Part 1 - Create Directory Structure

In this part you will create directory structure for storing event emitter application

- __ 1. Open **Command Prompt** from the **Start Menu**.
- __ 2. Switch to the **Workspace** directory

```
cd c:\workspace
```

Note: If the directory doesn't exist, create it by using md [c:\workspace](#) and then switch to it by executing the above command.

- __ 3. Create a directory named "promises" by using following command

```
md promises
```

- __ 4. Switch to the newly created directory

```
cd promises
```

Part 2 - Create file_promise.js Module

In this part you will create file_promise.js module. You will create a custom readFileAsync function which will utilize a Promise object to make readFile function asynchronous without using callbacks.

- __ 1. Copy **input.txt** from **c:\workspace\callbacks\input.txt** to **c:\workspace\promises\input.txt**
- __ 2. Use a text editor to create a new file named **file_promise.js** under **promises** directory.
- __ 3. Enter following code

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

Note: The above statement includes the **fs** module.

- __ 4. Below the above code, add following code

```
function readFileAsync (file) {
  return new Promise(function (resolve, reject) {
    fs.readFile(file, function (err, data) {
      if (err) // rejects the promise with "err" as the reason
        return reject(err);
      else // fulfills the promise with "data" as the value
        resolve(data);
    });
  });
}
```

Note: The custom `readFileAsync` function accepts file name to read as an argument and returns a `Promise` object to the `readFile` function. `readFile` function uses a callback function with error as first argument and data as second argument. Error argument is returned by the `reject` callback function of the `Promise` object and file contents are returned by the `resolve` callback function of the `Promise` object.

__5. Below the above function, add following code

```
var promise = readFileAsync('input.txt');
```

__6. The above statement calls the custom `readFileAsync` function and receives the `promise` object.

__7. Below the above statement, add the following code

```
promise.then(
  function(data) {
    console.log("Data is: \n" + data);
  })
  .catch(function(err) {
    console.log(err);
  })
);
```

The **then** function, of `promise` object, makes the actual `readFile` function call and displays the data on the console. The **catch** displays error message on the console if there's any issue while accessing the file.

__8. Save the file and close the text editor.

__9. In **Command Prompt** window, execute the `file_promise.js` file.

```
node file_promise.js
```

Notice the result looks like this

```
Data is:  
Line 1  
Line 2  
Line 3
```

Part 3 - Use Bluebird Module to Automatically Promisify readFile Function

In this part you will use Bluebird module to automatically promisify the readFile function. In fact, it will promisify the entire **fs** module / library. Promisifying a module, automatically generates the async function calls for the existing callback functions. It will reduce the overall effort required to utilize the Promise object.

__1. In the **Command Prompt**, ensure you are under **promises** directory

```
cd c:\workspace\promises
```

__2. Run following command to install Bluebird module

```
npm install bluebird@3.5.0
```

__3. Verify node_modules directory has been generated and it contains bluebird module.

```
dir node_modules
```

Notice there's bluebird directory under node_modules directory

__4. In the promises directory, duplicate **filePromise.js** as **file_bluebird.js**

__5. Below the existing `require('fs')` statement, add the following code

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

Note: This code imports the bluebird module.

__6. Delete the existing `readFileAsync` function.

Note: You are deleting the entire `readFileAsync` function definition so that bluebird can be used to promisify the `readFile` function.

__7. Below the existing **require** statements, add following code

```
var fs = bluebird.promisifyAll(fs);
```

Note: The above line uses bluebird module to promisify all callback-based functions in the fs module. It will automatically generate readFileAsync function which uses the promise object to make asynchronous function call.

__ 8. Locate the statement **var promise = readFileAsync('input.txt');** and change it to:

```
var promise = fs.readFileAsync('input.txt');
```

Note: Don't forget to add **fs.** before readFileAsync. Previously, you called the custom readFileAsync method. Whereas, this time you have promisified the fs module's readFile method. That's why fs. is required before readFileAsync.

__ 9. Save the file and close the text editor.

__ 10. In **Command Prompt** window, execute the file _promise.js file.

```
node file_bluebird.js
```

Notice the result looks like this, which is same as manual promise-based implementation performed in the previous part of this lab

```
Data is:
Line 1
Line 2
Line 3
```

Part 4 - Using Various Bluebird's Methods

In this part you will utilize Bluebird's all and map methods to read multiple files asynchronously.

__ 1. In the **Command Prompt**, ensure you are under **promises** directory

```
cd c:\workspace\promises
```

__ 2. Using a text editor, create simple text files with following names and contents under the promises directory

```
file name: input1.txt
```

```
contents: Hello
```

```
file name: input2.txt
```

```
contents: World
```

__3. Using a text editor, create a file named multi_file_all.js

__4. Enter following code

```
var Promise = require('bluebird');
```

Note: This statement includes the Bluebird module.

__5. Below the above statement, add following code

```
var fs = Promise.promisifyAll(require('fs'));
```

This statement promisifies the fs module callback-based methods.

__6. Add following function

```
function getFile(index) {  
  var filePath = __dirname + "/input" + index + ".txt";  
  return fs.readFileAsync(filePath);  
}
```

The function returns a file, either input1.txt or input2.txt depending on index value, asynchronously and returns its promise object.

__7. Below the above function, add following code

```
function getAllFiles() {  
  var promises = [];  
  // load all Files in parallel  
  for (var i = 0; i <= 1; i++) {  
    promises.push(getFile(i+1));  
  }  
  // return promise that is resolved when all Files are done loading  
  return Promise.all(promises);  
}
```

This function creates a promises array, reads and stores promise object of each file in the array, then all method returns a promise object which is resolved when all files are loaded.

__8. Below the above function, add following code

```

getAllFiles().then(function(fileArray) {
  // you have an array of File data in fileArray
  console.log("input1.txt " + fileArray[0]);
  console.log("input2.txt " + fileArray[1]);
})
.catch(function(err) {
  // an error occurred
  console.log(err);
});

```

The then method is called when all the files are read and displays the contents. If there is any error while reading the files the catch method is called.

__ 9. Save the file and close the text editor.

__ 10. Run following command in Command Prompt window

```
node multi_file_all.js
```

Notice it displays the result:

input1.txt Hello

input2.txt World

__ 11. Duplicate multi_file_all.js as multi_file_map.js

__ 12. Open multi_file_map.js in a text editor

__ 13. Delete getFile method

__ 14. Change the getAllFiles method as shown below

```

function getAllFiles() {
  return Promise.map(['input1.txt', 'input2.txt'], function(fileName) {
    return fs.readFileAsync(fileName);
  });
}

```

Note: This getAllFiles method is much shorter than the previous implementation where you used the all method. It returns a promise object for each file specified in the array.

__ 15. Save the file and close the text editor.

__ 16. Run following command in Command Prompt window

```
node multi_file_map.js
```

Notice it displays the result:

input1.txt Hello

input2.txt World

Part 5 - Create sumAsync Function to Make Asynchronous Function Call

In the previous part, you utilized a built-in `readFile` function along with promise object to make asynchronous function call. In this part, you will convert a custom `sum` function to use promises to execute it asynchronously without utilizing callbacks.

__ 1. Copy **sum.js** file from **c:\workspace\callbacks\sum.js** to **c:\workspace\promises\sum_promise.js**

__ 2. In a text editor open the file file

c:\workspace\promises\sum_promise.js

__ 3. Below the existing `sum` function definition, add following code

```
function sumAsync (a, b) {  
  return new Promise(function(resolve, reject) {  
    sum(a, b, function(err, data) {  
      if(err)  
        return reject(err);  
      else  
        resolve(data);  
    });  
  });  
}
```

Note: It's a custom `sumAsync` function which accepts the numbers to add up as arguments. The function returns a promise object to the `sum` callback-based function call. **resolve** callback function is called when data is valid and **reject** callback function is called when data is invalid.

__ 4. Below the existing `module.exports` statement, add following code

```
module.exports.sumAsync = sumAsync;
```

__ 5. Save the file and close the text editor.

__ 6. Using a text editor, create a new file named **main_promise.js** under **promises** directory.

__7. Add following code to the file

```
var sum = require('./sum_promise');
```

Note: This line imports the custom sum_promise module.

__8. Below the above line, add following code

```
var promise = sum.sumAsync(5,6);
```

This statement makes sumAsync function call and receives the promise object.

__9. Below the above statement, add following code

```
promise.then(  
  function(data) {  
    console.log("Sum is: \n" + data);  
  })  
  .catch(function(err) {  
    console.log(err);  
  })  
);
```

These lines call **then** function of the promise object. The callback function is called when data is processed successfully. The **catch** function is called when negative values are passed to the function or when there's any other error.

__10. Save the file and close the text editor.

__11. In Command Prompt window, run the following command to execute main_promise.js

```
node main_promise.js
```

Notice the result is

Sum is:

11

Part 6 - Use Bluebird Module to Promisify the Custom sum Function

In this part you will use Bluebird module to promisify the custom sum function. It will automatically generate the sumAsync function which will use promise object to make asynchronous function call.

__1. Under **promises** directory, duplicate **sum_promises.js** as **sum_bluebird.js**

- __2. Open **sum_bluebird.js** in a text editor
- __3. Remove the entire `sumAsync` function definition.
- __4. Remove the following statement

```
module.exports.sumAsync = sumAsync
```

- __5. Save the file and close the text editor.
- __6. Under `promises` directory, duplicate `main_promise.js` as `main_bluebird.js`
- __7. Open `main_bluebird.js` in a text editor
- __8. Remove the existing **require** statement
- __9. Add following code to top of the file

```
var sum = require('./sum_bluebird');  
var bluebird = require('bluebird');
```

Note: The above code imports `sum_bluebird` and `bluebird` modules.

- __10. Below the above code, add following code

```
var sum = bluebird.promisifyAll(sum);
```

- __11. Save the file and close the text editor.
- __12. In Command Prompt window, run the following command to execute `main_promise.js`

```
node main_bluebird.js
```

Notice the result is

Sum is:

11

Part 7 - Review

In this lab you utilized Promises to make asynchronous function call. You also used Bluebird module to promisify existing callback functions to make them asynchronous.

Lab 9 - Basic Web Application Development

Node.js comes with a core module called **http** that provides the foundation for a web application. You can build small scale applications with it. For more complex applications you will need a framework like Express that builds on top of the http module. It's always good to get some experience with the http module even if you intend to use a framework.

In this lab we will build a simple RESTful web service that will provide access to a movie database. We will only use the http module for that. To keep things simple we will keep all the data in memory and not use a real database.

Part 1 - Get Started

- ___ 1. In the **C:\LabFiles** folder create a folder called **movies**.
- ___ 2. In the **movies** folder create a file called **index.js**.
- ___ 3. In **index.js** add these lines. We will use this to verify that our basic web server is working.

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

var server = http.createServer(function(req, res) {
    res.write("Hello World\n");
    res.end();
});

server.listen(3000);
```

- ___ 4. Save changes.

Part 2 - Test

- ___ 1. In the command prompt window switch directory to **C:\LabFiles\movies**.
- ___ 2. Run this command to launch the application.

```
node index.js
```

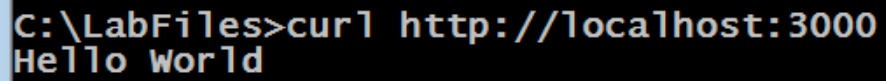
- ___ 3. Click **Allow Access** if a Windows Security Alert open.
- ___ 4. Open a new command prompt window.

__5. Switch to the **C:\LabFiles** folder.

__6. Run the **curl** command to send a HTTP request to the application.

```
curl http://localhost:3000
```

__7. Verify that you get the correct response.



```
C:\LabFiles>curl http://localhost:3000
Hello World
```

Great! Now that our basic web application is working we can now proceed to build our movie web service.

Part 3 - Build the In-Memory Database

__1. In the **index.js** file, below the line:

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

Add:

```
var movies = [
  {
    name:"Vampire in Brooklyn",
    releaseYear: 1995,
    tagLine:"A comic tale of horror and seduction."
  },
  {
    name:"Alien vs. Predator",
    releaseYear:2004,
    tagLine:"Whoever wins, we lose."
  },
  {
    name:"Chicken Run",
    releaseYear:2000,
    tagLine:"Escape or die frying."
  }
];
```

__2. Save changes.

Part 4 - Implement the GET Request

When there is a GET request we intend to return the full list of movies.

___1. In the request handler callback function delete these lines:

```
res.write("Hello World\n");  
res.end();
```

In their place add the lines shown below in boldface:

```
var server = http.createServer(function(req, res) {  
    res.setHeader("Content-type", "application/json");  
  
    if (req.method === "GET") {  
        res.write(JSON.stringify(movies));  
        res.end();  
    }  
});
```

Note: The res.write() method takes a Buffer or a String as the first parameter. That is why we had to convert the movies JavaScript object into String.

___2. Save changes.

Part 5 - Test

___1. In the command prompt that is running your application hit Control+C to end the program.

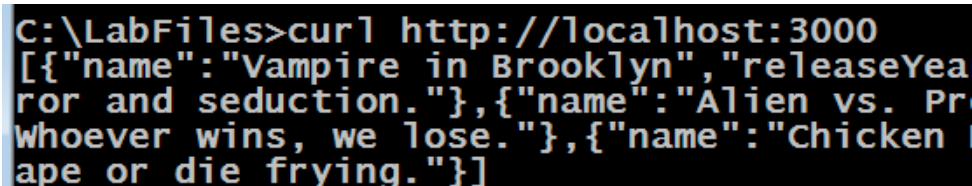
___2. Run the program again.

```
node index.js
```

___3. From the other command prompt window where you ran curl, send another request.

```
curl http://localhost:3000
```

___4. Verify that you see the full list of movies.



```
C:\LabFiles>curl http://localhost:3000  
[{"name":"Vampire in Brooklyn","releaseYear":1989,"description":"A vampire  
and seduction."}, {"name":"Alien vs. Predator",  
description":"Whoever wins, we lose."}, {"name":"Chicken  
ape or die frying."}]
```

Part 6 - Implement the POST Request

When a POST request is sent we intend to add a new movie to our database. JSON for the new movie will be in the request body. We can access the request body using the same Stream API that we used to read from file. This is because the request object implements the Readable interface. Let's get started.

___1. In the request handler function add the skeletal code for the POST request handling like shown in bold face below.

```
if (req.method === "GET") {  
  res.write(JSON.stringify(movies));  
  res.end();  
} else if (req.method === "POST") {  
  
}
```

___2. Within the if block for POST request handling add these lines in boldface.

```
} else if (req.method === "POST") {  
  var body = "";  
  
  req.on("data", function(chunk) {  
    body += chunk;  
  });  
  
  req.on("end", function() {  
    var newMovie = JSON.parse(body);  
    movies.push(newMovie);  
    res.end();  
  });  
}
```

Just as for files, the **data** event is fired by the request object as request body data is available to read. The **end** event is fired when the request body is fully read.

___3. Save changes.

Part 7 - Test

___1. Restart you application.

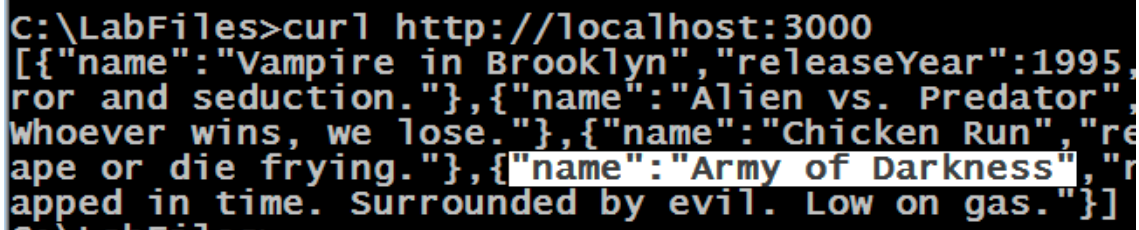
___2. We have data for a new movie available in **C:\LabFiles\movie.json**. Open the file in an editor and inspect it.

__3. From the command prompt where you ran curl, run this command to send a POST request. The request body will be supplied from the **movie.json** file.

```
curl http://localhost:3000 -d @movie.json
```

__4. Send a GET request and verify that the new movie was added.

```
curl http://localhost:3000
```



```
C:\LabFiles>curl http://localhost:3000
[{"name":"Vampire in Brooklyn","releaseYear":1995,"description":"horror and seduction."},{ "name":"Alien vs. Predator", "description":"whoever wins, we lose."},{ "name":"Chicken Run", "description":"ape or die frying."},{ "name":"Army of Darkness", "description":"appeared in time. Surrounded by evil. Low on gas."}]
```

__5. End your Node.js program by hitting Control+C.

__6. Close all.

Part 8 - Review

In this lab we used the http core more to build a very simple web service. We wrote our own code for basic routing based on the HTTP method.

Lab 10 - Debugging a Node.js Application

In this lab we will learn the basics of debugging a Node.js application using the **node-inspector** module.

This lab requires a working Internet connection since you will need to install the node-inspector module.

The node-inspector module uses Chrome's built-in debugger. You will need to have Chrome pre-installed.

Part 1 - Install node-inspector

__1. In a command prompt window switch directory to **C:\LabFiles**.

```
cd C:\LabFiles
```

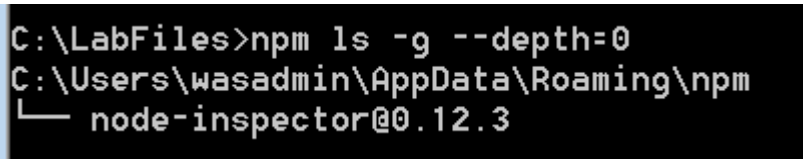
__2. Run this command to install the module.

```
npm install -g node-inspector@0.12.3
```

There will be a few errors but the module should install in the end.

__3. Verify that the module was installed by running this command.

```
npm ls -g --depth=0
```



```
C:\LabFiles>npm ls -g --depth=0
C:\Users\wasadmin\AppData\Roaming\npm
└─ node-inspector@0.12.3
```

If you do not see node-inspector installed try installing it again.

Part 2 - Debug a Program

We will now debug the movie web service that we had developed in an earlier lab.

__1. In a command prompt window switch to the C:\LabFiles\movies directory.

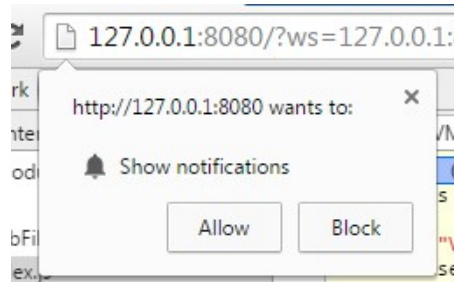
```
cd C:\LabFiles\movies
```

__2. Run this command to start debugging the program.

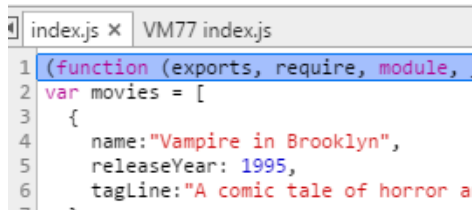
```
node-debug index.js
```


Troubleshooting. If the "node-debug is not recognized as an internal or external command" error message is displayed then add the following to the variable PATH in the Environment Variables. C:\Users\<<USER>>\AppData\Roaming\npm

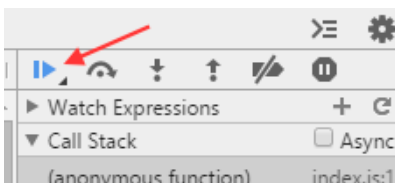
__3. That command will launch Chrome. You will get this prompt about notification. Click **Allow**.



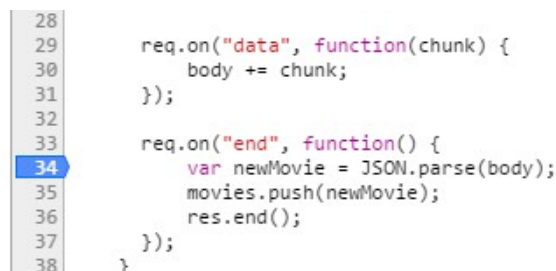
__4. By default the program will halt in the first line.



__5. Click the resume button  to continue.



__6. Add a breakpoint as shown below.



__7. Open a new command prompt window and switch to C:\LabFiles.

```
cd C:\LabFiles
```

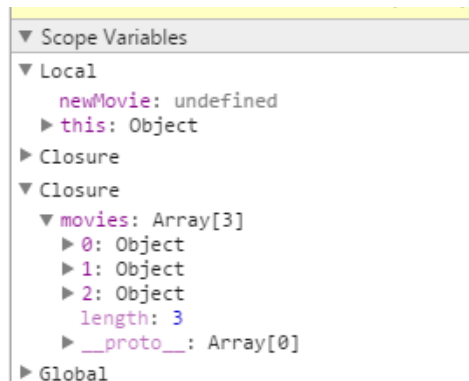
__8. Run this command to send a POST request.

```
curl http://localhost:3000 -d @movie.json
```

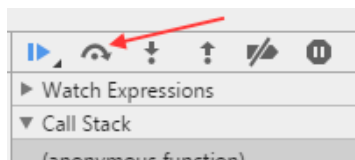
__9. The debugger will hit the break point and pause.

```
31     });  
32  
33     req.on("end", function() {  
34         var newMovie = JSON.parse(body);  
35         movies.push(newMovie);  
36         res.end();  
37     });
```

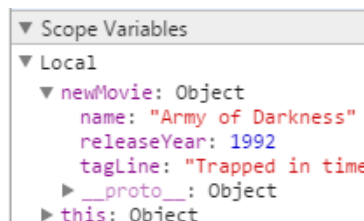
__10. On the right hand side expand **Closure** to locate the **movies** variable. There should be three movies there at this point.



__11. Click the step over button  to move to the next line.

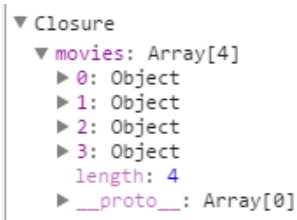



__12. Expand **Local** and inspect the **newMovie** variable.



__13. Click the step over button  to move to the next line.

__14. At this point there should be 4 items in the movies array.



__15. Click the resume button  to continue.

__16. Close the Chrome window.

__17. In the command prompt where you ran node-debug hit Control+C to end the program.

__18. Close all.

Part 3 - Review

In this lab we learned to debug a Node.js program using the node-inspector module. This module uses the debugger built into the Chrome web browser.

Lab 11 - Introduction to Unit Testing

Unit testing can help you identify defects before your code reaches a test environment. Automated unit testing encourages developers to perform more unit testing.

In this lab we will test a Fibonacci sequence generator module. We will use mocha and chai for this.

Part 1 - Create the Module

___1. In **C:\LabFiles** folder create a directory called **fib**.

___2. Inside the **C:\LabFiles\fib** folder create **package.json** file with this content.

```
{
  "name": "fib",
  "version": "1.0.0"
}
```

___3. Save changes.

___4. In the same folder create the main code file **index.js** and add this code.

```
module.exports.calculate = function(n) {
  var a = 0, b = 1, f = 1;

  for (var i = 2; i <= n; i++) {
    f = a + b;
    a = b;
    b = f;
  }

  return f;
};
```

___5. Save changes.

Our module is a reusable library. It has no entry point that we can run to verify if our code is working or not. We will need to write automated unit test script to do that.

Part 2 - Configure the Module for Testing

Let's install the **mocha** and **chai** modules. They are only needed during development. A user of our fib module will not need to depend on them. As a result we setup a development time dependency.

__1. Open a command prompt window and change directory to C:\LabFiles\fib.

```
cd C:\LabFiles\fib
```

__2. Enter these commands to install the required modules.

```
npm install mocha@2.3.3 --save-dev
```

```
npm install chai@3.3.0 --save-dev
```

__3. Verify that package.json was altered to add the development time dependencies.

```
{
  "name": "fib",
  "version": "1.0.0",
  "devDependencies": {
    "chai": "^3.3.0",
    "mocha": "^2.3.3"
  }
}
```

__4. We will now setup mocha as the test runner. Add the script section to package.json as shown in boldface below.

```
{
  "name": "fib",
  "version": "1.0.0",
  "scripts": {
    "test": "mocha test"
  },
  "devDependencies": {
    "chai": "^3.3.0",
    "mocha": "^2.3.3"
  }
}
```

Basically we configure "mocha test" as the command that npm will run to initiate test. The argument to mocha ("test" in this case) identifies the folder where the test scripts are kept.

__5. Save changes.

Part 3 - Write Test Scripts

__1. Inside **C:\Labfiles\fib** create a subdirectory called **test**. This is where all test scripts will go.

__2. Inside the test folder create a file called **fibonacci_tester.js**. We will write our test scripts here.

__3. In **fibonacci_tester.js** add these lines to import the necessary modules.

```
var fib = require("../..//fib");  
var chai = require("chai");
```

Note we have to use the path **"../..//fib"** to load our module.

We will now write the test scripts. A test suite is a function registered with the **describe()** function. It is a collection of tests. From this function we register one or more test functions using the **it()** function.

__4. Write three test functions like this.

```
describe('Main test suite', function() {  
  it('Edge case 1', function () {  
    chai.assert.equal(fib.calculate(1), 1, "Should be 1");  
  });  
  it('Edge case 2', function () {  
    chai.assert.equal(fib.calculate(2), 1, "Should be 1");  
  });  
  it('Regular case', function () {  
    chai.assert.equal(fib.calculate(10), 55, "Should be 55");  
  });  
});
```

__5. Save changes.

Part 4 - Run Unit Test

__1. From a command prompt window change directory to **C:\LabFiles\fib**.

```
cd C:\LabFiles\fib
```

__2. Enter this command to run the tests.

```
npm test
```

__3. Make sure that all tests pass.

```
Main test suite
  ✓ Edge case 1
  ✓ Edge case 2
  ✓ Regular case

3 passing (0ms)
```

Part 5 - Test for Failure

A test can fail because you have a bug in your module code or the test script itself is buggy. We will now introduce a bug in our module code and observe how mocha displays failed tests.

__1. In the index.js file change the return statement as shown in boldface below.

```
return f + 2;
```

__2. Save changes.

__3. Run the test again.

```
Main test suite
  1) Edge case 1
  2) Edge case 2
  3) Regular case

0 passing (0ms)
3 failing

1) Main test suite Edge case 1:
  AssertionError: Should be 1: expected 3 to equal 1
    at Context.<anonymous> (C:\LabFiles\fib\test\fibonacci.js:10:12)
```

__4. Verify that all three tests failed. Also read through the error messages to find out how exactly did the tests fail.

__5. Fix the problem with the code and run the test again to make sure all is well.

__6. Close the editor.

__7. Leave the command prompt open.

Part 6 - Review

In this lab we learned how to write basic test scripts using mocha. We use chai to write our assertion logic. Once a module is configured for test one can easily execute the test scripts by running **npm test**.

Lab 12 - Logging with Morgan

In this lab, we will learn how to use **morgan**, the HTTP request logger you can use in your applications running on Node.js.

Part 1 - Create the Working Directory

__1. Open a command prompt window and create a working directory for this lab:

```
mkdir C:\Works\morgan
```

__2. Change to the lab's working directory you just created.

```
cd C:\Works\morgan
```

__3. Enter the following command to set the title of the command prompt window:

```
title morgan
```

Part 2 - Install morgan from the Internet

Note: This step requires direct access to the Internet. If you don't have a working Internet connection, proceed to the **Install morgan from the Archived Local Package** lab part.

__1. Run this command to locally install the *morgan* module version 1.7.0.

```
npm install morgan@1.7.0
```

Wait for the download and installation process to complete.

Note: We install version 1.7.0 of the module as this version was tested during the lab creation process.

Part 3 - Install morgan from the Archived Local Package

__1. Make sure you are in the "**morgan**" command prompt window and in the *C:\Works\morgan* directory.

__2. Enter the following command:

```
copy c:\LabFiles\node_modulesA\morgan@1.7.0.zip .
```

Note: The *morgan@1.7.0.zip* file contains the previously downloaded *morgan* module.

__3. Open a file browser to **C:\Works\morgan**.

__4. Extract the **morgan@1.7.0.zip** in this folder and make sure that you will have the following folder structure:

```
C:\Works\morgan\node_modules\morgan
```

__5. Back in the command prompt enter this command to make sure you extracted the files correctly. If you get an error then fix the location of the extracted file.

```
dir node_modules\morgan
```

Part 4 - Verify Module Installation

__1. Make sure you are in the "**morgan**" command prompt window and in the *C:\Works\morgan* directory.

__2. Enter the following command to make sure the *morgan* module was properly installed:

```
npm ls
```

You should see the following output:

```
C:\Works\morgan
├─┬ morgan@1.7.0
│   ├── basic-auth@1.0.4
│   ├── debug@2.2.0
│   │   └── ms@0.7.1
│   ├── depd@1.1.0
│   ├── on-finished@2.3.0
│   ├── ee-first@1.1.1
│   └── on-headers@1.0.1
```

Notice that *morgan* depends on the *debug* module [<https://www.npmjs.com/package/debug>] which is a debugging utility used for logging by Node.js applications.

Part 5 - Set up and Run a Simple HTTP Server

Let's see what options we have out of the box for logging in our Node.js applications.

__1. Enter the following command:

```
copy c:\LabFiles\simpleHTTPServer.js .
```

__2. Start your text editor and open *C:\Works\morgan\simpleHTTPServer.js* you just copied over to your working folder.

__3. Review its content shown below for your reference.

```
var http = require('http');
//var morgan = require ('morgan');
//var logger = morgan('combined');

var port = 12345;

http.createServer(function(req, res){
    var timeIn = new Date();
    var timeStamp = timeIn.getHours() + ":" + timeIn.getMinutes() + ":" + timeIn.getSeconds() + "." +
timeIn.getMilliseconds();timeIn.getMinutes() + ":" + timeIn.getSeconds() + "." +
timeIn.getMilliseconds();
    var logMsg = timeStamp + ' Received HTTP request: [' + req.method + ']' [' + req.url + ']';
    console.log(logMsg);

    // logger(req, res, function (err) {}));

    res.setHeader('content-type', 'text/plain')
    res.end(logMsg);
}).listen(port, function(){
    console.log('The server is listening on port ' + port + ' ...');
});
```

You will notice that logging is done with the *console.log* statements and the HTTP request log message, represented by the *logMsg* variable in our case, requires a bit of manual work.

__4. Keep the text editor running and the *simpleHTTPServer.js* file opened in the text editor.

__5. Switch back to our command prompt window and enter the following command (you should be in the *C:\Works\morgan* directory):

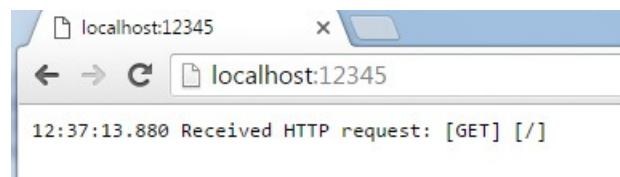
```
node simpleHTTPServer.js
```

You should see the following diagnostic message printed in the command prompt window:

```
The server is listening on port...12345
```

__6. Open your browser and navigate to **http://localhost:12345**

You should see the following content rendered in the browser (your timestamp will be different):



__7. Keep the browser running.

__8. Switch back to the command prompt window.

There you should see the following output related to your request(s) from Chrome:

```
12:37:13.880 Received HTTP request: [GET] [/]  
12:37:13.913 Received HTTP request: [GET] [/favicon.ico]
```

Note: Browsers automatically issue an HTTP request for the *favicon.ico* image to display it, if it is present, in the navigation bar next to the site's URL.

As you can see, we were able to capture and print some basic details of the incoming HTTP requests.

__9. Shutdown the web server by pressing **Ctrl-C**

Part 6 - Add morgan Logging

Now let's add morgan logging to our simple HTTP server.

__1. In the text editor open on **simpleHTTPServer.js**, save the **simpleHTTPServer.js** file as **consoleMorganLogger.js**

__2. In the *consoleMorganLogger.js* file, uncomment the following lines related to *morgan*.

```
//var morgan = require ('morgan');  
//var logger = morgan('combined');  
...  
//logger(req, res, function (err) {} );
```

By uncommenting the above lines, we enable *morgan's* logging as per Standard Apache *combined* log output format; to simplify code, we ignore any processing errors that may occur.

__3. Save the changes.

__4. Switch back to the command prompt window and enter the following command (you should be in the *C:\Works\morgan* directory):

```
node consoleMorganLogger.js
```

You should see the following diagnostic message:

```
The server is listening on port...12345
```

__5. Open your browser and refresh the resource identified by the **http://localhost:12345** URL.

__6. Switch to the command prompt window and review the log messages related to the browser HTTP requests.

Note A: Your timestamps in the output will be different.

Note B: The log messages emitted by *morgan* alongside our custom *console.log* based messages are shown below in **bold**.

```
13:4:46.564 Received HTTP request: [GET] [/]

::1 - - [07/Jul/2016:17:04:46 +0000] "GET / HTTP/1.1" 200 - "-" "Mozilla/5.0
(Windows NT 6.1) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/51.0.2704.103 Safari/537.36"

13:4:46.602 Received HTTP request: [GET] [/favicon.ico]

::1 - - [07/Jul/2016:17:04:46 +0000] "GET /favicon.ico HTTP/1.1" 200 -
"http://localhost:12345/" "Mozilla/5.0 (Windows N
T 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.103
Safari/537.36"
```

As you see, we have much more useful information about the client requests and the server response.

__7. Shutdown the web server by pressing **Ctrl-C**

Part 7 - Logging to a File

Now let's see what code changes are required to send our application's logging output to a file rather than to a console.

__1. In the text editor open on **consoleMorganLogger.js**, save the **consoleMorganLogger.js** file as **fileMorganLogger.js**

__2. In the *fileMorganLogger.js* file, do the following changes:

Replace this line:

```
var logger = morgan('combined');
```

with this code:

```
var fs = require('fs');
var accessLogStream = fs.createWriteStream(__dirname + '/access.log', {flags: 'a'})
var logger = morgan('combined', {stream: accessLogStream});
```

So that the top of the file should look as follows:

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

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

var fs = require('fs');
var accessLogStream = fs.createWriteStream(__dirname + '/access.log', {flags: 'a'})
var logger = morgan('combined', {stream: accessLogStream});

var port = 12345;
```

The new code does the following:

- Initialize access to the local file system from our application
- Create a file named *access.log* in the current directory and open it in *append* mode when the first HTTP request comes through.

___3. Save the changes.

___4. Switch back to the command prompt window and enter the following command (you should be in the *C:\Works\morgan* directory):

```
node fileMorganLogger.js
```

You should see the following diagnostic message:

```
The server is listening on port...12345
```

___5. Open your browser and refresh the **http://localhost:12345** URL.

___6. Switch to the command prompt window where you started the web server.

You should only see the log messages coming from our *console.log* statements.

___7. Using your text editor, open the **C:\Works\morgan\access.log** file.

You should see that the log messages generated by *morgan* are diligently send to this log file.

___8. Shutdown the web server by pressing **Ctrl-C**

Part 8 - Optional Activity 1

Try to experiment with other pre-defined log message format, like

```
common
dev
short
```

Any one of those values should go as a parameter to the *morgan* function, e.g.:

```
var logger = morgan('dev');
```

Don't forget to save the changes in the file before restarting the web server !

When done, shut down the web server by pressing **Ctrl-C**

Part 9 - Optional Activity 2

Our simple HTTP server is pretty dumb and happily informs the clients that everything is hunky-dory (OK), where in case of the *favicon.ico* asset we should probably return HTTP status code 404 - Not found.

If you are interested in extending the rather limited functionality of our web server, try the following:

Add the **res.statusCode = 404;** line for HTTP requests asking for the */favicon.ico* resource.

Note: The requested asset (*/favicon.ico*, in our case) is captured in the **req.url** request property.

When done, shut down the web server by pressing **Ctrl-C**

Part 10 - Lab Session Clean-Up

We are almost done in our lab.

- __ 1. Close Chrome.
- __ 2. Close your text editor.
- __ 3. Shutdown the web server by pressing **Ctrl-C**
- __ 4. Close the command prompt window.

Part 11 - Review

In this lab we demonstrated the logging capabilities offered by the *morgan* module.

Lab 13 - Web Service Using Express

Express is a web application framework that builds on top of the http core module. In this lab we will build bibliography web service using Express. The web service will have these interfaces.

- GET /books/*ISBN* – Return a book by the ISBN number.
- PUT /books/*ISBN* – Create or update a book.

Part 1 - Create the Module

__1. In the **C:\LabFiles** folder create a directory called **biblio**.

__2. In the **biblio** folder create **package.json** like this.

```
{  
  "name": "biblio",  
  "version": "1.0.0"  
}
```

__3. Save changes.

We will now install the express module.

__4. In a command prompt window change directory to C:\LabFiles\biblio.

```
cd C:\LabFiles\biblio
```

__5. Install express and body-parser. We need the body-parser module to be able to read the POST request body.

```
npm install express@4.13.3 --save
```

```
npm install body-parser@1.14.0 --save
```

Part 2 - Create Business Logic Sub-Module

We will now create a submodule called `data_access`. It will have the core business logic that completely removed from the web. For example, the same business logic can be used from a command line application.

__1. In C:\LabFiles\biblio create a JavaScript file called **data_access.js**.

__2. Add the following code to that file.

```
var books = {}; //In-memory database

module.exports.findBook = function(isbn) {
    return books[isbn];
};

module.exports.updateBook = function(isbn, book) {
    books[isbn] = book;
};
```

__3. Save changes.

Part 3 - Create the Web Service

__1. In C:\LabFiles\biblio create a JavaScript file called **index.js**.

__2. In that file add this line to import data_access, express and body-parser modules.

```
var express = require('express');
var bodyParser = require('body-parser');
var dao = require("../data_access");
```

__3. Obtain the Express application object.

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

First thing we will do is register the body parser middleware. It will parse all request bodies when content type is JSON.

__4. Add this code.

```
app.use(bodyParser.json()); //Parse JSON body
```


__ 5. Add the route for GET request.

```
app.get("/books/:isbn", function(req, res) {  
  var book = dao.findBook(req.params.isbn);  
  
  if (book === undefined) {  
    res.statusCode = 404;  
    res.end();  
  } else {  
    res.send(book);  
  }  
});
```

Note how we retrieve the isbn path parameter using req.params.isbn.

Also note that when you supply an object to res.send() it will automatically set the Content-type header to application/json.

__ 6. Add the route for PUT request.

```
app.put("/books/:isbn", function(req, res) {  
  if (req.params.isbn === undefined || req.body === undefined) {  
    res.statusCode = 500;  
    res.end();  
  
    return;  
  }  
  
  dao.updateBook(req.params.isbn, req.body);  
  res.end();  
});
```

Note how we retrieve the request body as a JavaScript object as req.body. This is possible because the body-parser middleware has read and parsed the request body and saved the object as the body property of the request.

__ 7. Add this line to listen on port 3000.

```
app.listen(3000);
```

__ 8. Save changes.

Part 4 - Test

__ 1. From a command prompt window change directory C:\LabFiles\biblio.

```
cd C:\LabFiles\biblio
```

__2. Run your program.

```
node index.js
```

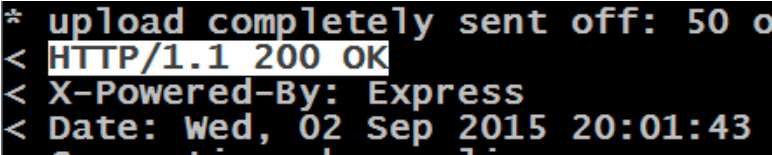
__3. You may request to give access to Node.js. Do it.

__4. From another command prompt window go to C:\LabFiles.

```
cd C:\LabFiles
```

__5. Use curl to send a PUT request. We will get the body data from book1.json. The command is broken up in two lines. But should be entered as one line.

```
curl -v -X PUT -H "Content-Type: application/json"  
-d @book1.json http://localhost:3000/books/123
```

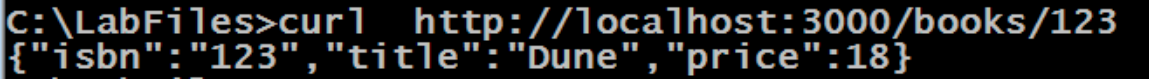


```
* upload completely sent off: 50 o  
< HTTP/1.1 200 OK  
< X-Powered-By: Express  
< Date: Wed, 02 Sep 2015 20:01:43
```

__6. Verify that status code 200 is returned.

__7. Send a GET request to retrieve the book back.

```
curl http://localhost:3000/books/123
```



```
C:\LabFiles>curl http://localhost:3000/books/123  
{"isbn":"123","title":"Dune","price":18}
```

__8. Verify that you get the book data back.

__9. Send a request for a non-existent ISBN number and make sure that you get 404 status back.

```
curl -v http://localhost:3000/books/abc
```

__10. Press CTRL+C to stop the running process.

__11. Close all.

Part 5 - Review

In this lab we used the Express framework to create a basic web service. We saw how routing works with Express. We also got to use a middleware to parse the request body.

We isolated the core business logic in its own submodule. Among other advantages, this lets us easily unit test the business logic layer.

Lab 14 - Using MongoDB

Previously in our biblio web service we stored the data in memory. In this lab we will convert the code to store data in MongoDB.

Part 1 - Run MongoDB

- __ 1. Close all open windows and commands prompts.
- __ 2. Verify if **C:\data\db** folder exist, if not then create the folder.
- __ 3. Open a command prompt window.
- __ 4. Change directory to where MongoDB binary is located.

```
cd C:\Software\mongodb\bin
```

- __ 5. Start MongoDB server by running:

```
mongod
```

- __ 6. If a Windows Security Alert open, click **Allow Access**.

```
e Pack 1 ) BOOST_LIB_VERSION=1_49
2015-09-03T15:12:57.935-0400 I CONTROL [initandlisten] allocator: tcmalloc
2015-09-03T15:12:57.936-0400 I CONTROL [initandlisten] options: {}
2015-09-03T15:12:57.943-0400 I NETWORK [initandlisten] waiting for connections
on port 27017
```

- __ 7. If all goes well you will see the "waiting for connections on port 27017" message.

Troubleshooting

MongoDB may fail to start.

```
2015-09-03T15:08:21.537-0400 I CONTROL Hotfix KB2731284 or 1
installed, will zero-out data files
2015-09-03T15:08:21.537-0400 W - [initandlisten] Detecte
n - C:\data\db\mongod.lock is not empty.
2015-09-03T15:08:21.537-0400 I STORAGE [initandlisten] *****
Unclean shutdown detected.
Please visit http://dochub.mongodb.org/core/repair for recover
*****
2015-09-03T15:08:21.538-0400 I STORAGE [initandlisten] excep
en: 12596 old lock file, terminating
2015-09-03T15:08:21.538-0400 I CONTROL [initandlisten] dbexf
```

If you see the **C:\data\db\mongod.lock** is not empty message, delete the file and try starting again.

Part 2 - Install mongodb Module

We need the mongodb module to connect to a MongoDB database.

- ___1. Open a new command prompt window.
- ___2. Change directory to C:\LabFiles\biblio.

```
cd C:\LabFiles\biblio
```

- ___3. Install mongodb module.

```
npm install mongodb@2.2.25 --save
```

Part 3 - Open Database Connection

We will now modify C:\LabFiles\biblio\data_access.js to use MongoDB. First we will open a connection to the database. The connection object is actually a connection pool. For a web application we should keep this pool open for the entire duration of the application.

- ___1. Open C:\LabFiles\biblio\data_access.js in an editor.
- ___2. Delete the following line. The new implementation will be quite different.

```
var books = {}; //In-memory database
```

- ___3. At the very top of the file add:

```
var mongodb = require("mongodb");
var dbPool; //Connection pool
var url = "mongodb://localhost:27017/biblio_db";

mongodb.MongoClient.connect(url, function(err, db) {
    if (err === null) {
        dbPool = db; //Open the pool
    }
});
```

Basically, the pool will be opened when the module is loaded for the first time.

- ___4. Save changes.

Part 4 - Implement the findBook Method

MongoDB returns a query result asynchronously. That means our **findBook** method also has to become asynchronous.

__1. Delete the body of the findBook method so that it looks like this.

```
module.exports.findBook = function(isbn) {  
  
}
```

__2. Add a callback parameter to the method as shown in bold face below. We will call this callback function after MongoDB returns data.

```
module.exports.findBook = function(isbn, callback) {  
  
}
```

__3. In the body of the method add the lines shown in boldface below.

```
module.exports.findBook = function(isbn, callback) {  
  var col = dbPool.collection("books");  
  
  col.find({isbn: isbn})  
    .toArray(function(err, books) {  
      if (err === null && books.length > 0) {  
        callback(null, books[0]);  
      } else {  
        callback("Failed to find book");  
      }  
    });  
};
```

__4. Save changes.

Part 5 - Implement the updateBook Method

The updateBook method inserts or updates a book. This is equivalent to the upsert operation of MongoDB.

__1. Delete the body of the updateBook method and add the callback parameter as shown in bold face below.

```
module.exports.updateBook = function(isbn, book, callback) {  
  
}
```

__2. Within the body of the method add the lines shown in boldface below.

```
module.exports.updateBook = function(isbn, book, callback) {  
    var col = dbPool.collection("books");  
  
    col.update(  
        {isbn: isbn},  
        {$set: book},  
        {upsert: true},  
        callback);  
};
```

__3. Save changes.

Part 6 - Modify the Web Service

__1. Open C:\LabFiles\biblio\index.js in an editor.

__2. Modify the GET request handler as shown in boldface below.

```
app.get("/books/:isbn", function(req, res) {  
    dao.findBook(req.params.isbn, function(err, book) {  
        if (book !== undefined) {  
            //We have a book  
            res.send(book);  
        } else {  
            res.statusCode = 404;  
            res.end();  
        }  
    });  
});
```

Basically, the handler now has to register a callback function to receive the data back.

Important

Make sure that you don't call `res.end()` directly from the request handler function. Doing so will prematurely end the response document while we are still waiting for data back from MongoDB. The response document should be closed only from the callback that receives data back from MongoDB.

__3. Modify the PUT handler function as shown below.

```
app.put("/books/:isbn", function(req, res) {
  if (req.params.isbn === undefined || req.body === undefined) {
    res.statusCode = 500;
    res.end();

    return;
  }

  dao.updateBook(req.params.isbn, req.body, function(err) {
    if (err !== null) {
      res.statusCode = 500;
    }
    res.end();
  });
});
```

__4. Save changes.

Part 7 - Test

Testing will be same as before.

__1. From a command prompt window change directory C:\LabFiles\biblio.

```
cd C:\LabFiles\biblio
```

__2. Run your program.

```
node index.js
```

__3. From another command prompt window go to C:\LabFiles.

```
cd C:\LabFiles
```

__4. Use curl to send a PUT request. We will get the body data from book1.json. The command is broken up in two lines. But should be entered as one line.

```
curl -v -X PUT -H "Content-Type: application/json"
  -d @book1.json http://localhost:3000/books/123
```


__5. Verify that status code 200 is returned.

```
* upload completely sent off: 50 o
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Date: Wed, 02 Sep 2015 20:01:43
```

__6. Send a GET request to retrieve the book back.

```
curl http://localhost:3000/books/123
```

```
C:\LabFiles>curl http://localhost:3000/books/123
{"_id":"55f1865ad92abccd43165097","isbn":"123","title":"Dune","price":18}
```

__7. Verify that you get the book data back. Note that MongoDB has added a property called **_id**. This acts as kind of a primary key. By default only this property is indexed by MongoDB. We will learn about indexing later in this lab.

__8. Send a request for a non-existent ISBN number and make sure that you get 404 status back.

```
curl -v http://localhost:3000/books/abc
```

Part 8 - Create an Index

In this lab we have decided to query using the isbn property. But by default only the **_id** property is indexed by MongoDB. As a result a query by isbn will cause MongoDB to scan all documents in the collection. Having proper indices is key to high performance queries. So let's dig a little deeper into this.

__1. Open a command prompt window.

__2. Change directory to the **bin** folder of MongoDB.

```
cd C:\Software\mongodb\bin
```

__3. Run the MongoDB command line client.

```
mongo
```

__4. By default the client connects to the **test** database. Enter this command to switch to the **biblio_db** database.

```
use biblio_db
```

__5. Enter this command to do a query and make sure that you can see the document inserted earlier by the web service.

```
db.books.find({isbn: "123"})
```

```
> db.books.find({isbn: "123"})
{ "_id" : ObjectId("55f1865ad92abccd43165097"), "isbn" : "123", "title" : "Dune", "price" : 18 }
```

OK. So far so good. Now we will use the query explain tool to find out exactly how MongoDB is doing this query.

__6. Enter this command to get an explanation for the query.

```
db.books.explain().find({isbn: "123"})
```

__7. The most important property for us is **stage**. It should be set to **COLLSCAN**. That indicates that no index is used and the database has to perform full collection scan (go through each item in the collection) to look for the matching documents.

```
    "parsedQuery" : {
      "isbn" : {
        "$eq" : "123"
      }
    },
    "winningPlan" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "isbn" : {
          "$eq" :
        }
      },
      "direction" : "forward"
    }
```

Needless to say that this is not a very good situation. The query will be super slow if the collection has many documents. We will now define an index for the **isbn** property.

__8. Enter this command to index the isbn property.

```
db.books.createIndex( { "isbn" : 1 } )
```

```
> db.books.createIndex( { "isbn" : 1 } )
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 1,
  "numIndexesAfter" : 2,
  "ok" : 1
}
```

__9. Now run the explain tool again.

```
db.books.explain().find({isbn: "123"})
```

__10. Verify that the **stage** property is now set to **IXSCAN** indicating that an index scan was performed.



```
"winningPlan" : {
  "stage" : "FETCH",
  "inputStage" : {
    "stage" : "IXSCAN",
    "keyPattern" : {
      "isbn" : 1
    }
  }
}
```

__11. Enter **Control+D** to quit the client tool.

Part 9 - Use ES6 Arrow Function

Node.js 4.x has great support for ECMAScript 2015. We will now convert the callback functions in this lab into arrow functions. Arrow functions have two advantages over regular callback functions:

1. They require a little less typing.
2. The **this** keyword within an arrow function is same as the **this** of the enclosing function. This proves advantageous when the context of the outer function must be preserved in the callback.

We don't really have to use arrow functions in our simple application. But lets do so anyway just to get used to it.

__1. Open **data_access.js** in an editor.

__2. In the **findBook** exported method, locate the callback function supplied to **toArray()** as shown below.

```
col.find({isbn: isbn})
  .toArray(function(err, books) {
```

__3. Convert it to an arrow function as shown in bold face below.

```
col.find({isbn: isbn})
  .toArray((err, books) => {
```

__4. Save changes.

__5. Open **index.js**.

__6. Convert the callback supplied to **findBook()** as shown in bold face below.

```
dao.findBook(req.params.isbn, (err, book) => {
```

__7. Similarly, convert the callback supplied to **updateBook()**.

```
dao.updateBook(req.params.isbn, req.body, (err) => {
```

__8. Convert all the route request handler functions to arrow functions also. For example:

```
app.get("/books/:isbn", (req, res) => {
```

And

```
app.put("/books/:isbn", (req, res) => {
```

__9. Save changes.

Part 10 - Test

__1. End you web service application by hitting **Control+C**.

__2. Run it again.

```
node index.js
```

__3. Open a command and change to C:\LabFiles

```
cd C:\LabFiles
```

__4. Run this command from C:\LabFiles to verify all is working well.

```
curl http://localhost:3000/books/123
```

Part 11 - Clean Up

__1. End you web service application by hitting **Control+C**.

__2. End MongoDB by hitting **Control+C**.

__3. Close all.

Part 12 - Review

In this lab we learned to use MongoDB. A couple of key things to keep in mind:

1. Make sure that you are using connection pool. Do not close the pool. This will improve performance manyfold.
2. MongoDB returns data asynchronously. That means the interface to your data access layer will also become asynchronous. The web service will have to register a callback.
3. Make sure that you do not end the response document while you are still waiting for data from MongoDB.
4. Make sure that your queries use indices. Use the explain tool to verify that.

Lab 15 - Using the Jade Template Engine

A template engine comes handy when the server side needs to render dynamic HTML. Jade is a popular template engine for Node.js.

In this lab we will create a web page that will show a list of all books.

Biblio Database

ISBN	Title	Price
123	Dune	18
456	The Stranger on the Train	4.48

Part 1 - Implement the Data Access Logic

We will now write code that will return all books from the MongoDB database.

1. Open `C:\LabFiles\biblio\data_access.js` in an editor.
2. Add a new exported function like this.

```
module.exports.findAllBooks = function(isbn, callback) {  
  var col = dbPool.collection("books");  
  
  col.find()  
    .toArray((err, books) => {  
      if (err === null) {  
        callback(null, books);  
      } else {  
        callback("Failed to find books", undefined);  
      }  
    })  
};
```

3. Save changes.

Part 2 - Write Controller Logic

A controller will get the data from the data access layer and pass it to a Jade template which makes up the view layer. We will write that code now.

1. First we will install the jade package. In a command prompt window go to `C:\LabFiles\biblio`.

```
cd C:\LabFiles\biblio
```

__2. Run this command.

```
npm install jade@1.11.0 --save
```

We will now configure Jade as the template engine.

__3. Open **C:\LabFiles\biblio\index.js** in an editor.

__4. Below the line:

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

Add:

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

This configures Jade as the view engine and states that the views are located in the `./views` subfolder.

We will now create a new route for a GET request for `/books`. This will return a HTML page listing all books.

__5. Above the line:

```
app.listen(3000);
```

Add:

```
app.get("/books", (req, res) => {  
  dao.findAllBooks(req.params.isbn, (err, books) => {  
    if (books !== undefined) {  
      res.render("book_list", {  
        book_collection: books  
      });  
    } else {  
      res.statusCode = 500;  
      res.end();  
    }  
  });  
});
```

The key part of this code is the call to `res.render()`. It states that view should be rendered using the `book_list.jade` file. This also passes an object to the template. This object has a property called `book_collection` that has the array of books.

__6. Save changes.

Part 3 - Write the View Template

__1. In the **C:\LabFiles\biblio** folder create a subdirectory called **views**. We have configured this as the folder where the Jade templates are stored.

__2. In the **C:\LabFiles\biblio\views** folder create a file called **book_list.jade**.

__3. In **book_list.jade** add these lines to get things started.

```
html
  body
    h1 Biblio Database
```

Jade treats the first word in a line as a tag name. Also indentation signifies nesting. So the code above will generate the following HTML:

```
<html>
  <body>
    <h1>Biblio Database</h1>
  </body>
</html>
```

OK, that's great so far. Now lets create a table that will show all books.

__4. Add the lines below shown in bold face. Make sure that the table tag is at the same level of indentation as h1 (since they are siblings):

```
html
  body
    h1 Biblio Database
    table
      tr
        td ISBN
        td Title
        td Price
      each book in book_collection
        tr
          td #{book.isbn}
          td #{book.title}
          td #{book.price}
```

Note how we are iterating over the **book_collection** variable. This is supplied by the controller when the template is loaded. You can use tabs or spaces in the **book_list.jade** file both not both.

__5. Save changes.

Part 4 - Test

- ___ 1. Open a new command prompt window.
- ___ 2. Change directory to where MongoDB binary is located.

```
cd C:\Software\mongodb\bin
```

- ___ 3. Start MongoDB server by running:

```
mongod
```

- ___ 4. If a Windows Security Alert open, click **Allow Access**.

```
e Pack 1 J BOOST_LIB_VERSION=1_49
2015-09-03T15:12:57.935-0400 I CONTROL [initandlisten] allocator: tcmalloc
2015-09-03T15:12:57.936-0400 I CONTROL [initandlisten] options: {}
2015-09-03T15:12:57.943-0400 I NETWORK [initandlisten] waiting for connections
on port 27017
```

- ___ 5. If all goes well you will see the "waiting for connections on port 27017" message.

Troubleshooting

MongoDB may fail to start.

```
2015-09-03T15:08:21.537-0400 I CONTROL Hotfix KB2731284 or
installed, will zero-out data files
2015-09-03T15:08:21.537-0400 W - [initandlisten] Detecte
n - C:\data\db\mongod.lock is not empty.
2015-09-03T15:08:21.537-0400 I STORAGE [initandlisten] *****
Unclean shutdown detected.
Please visit http://dochub.mongodb.org/core/repair for recovery
*****
2015-09-03T15:08:21.538-0400 I STORAGE [initandlisten] excep
en: 12596 old lock file, terminating
2015-09-03T15:08:21.538-0400 I CONTROL [initandlisten] dbexi
```

If you see the **C:\data\db\mongod.lock** is not empty message, delete the file and try starting again.

- ___ 6. Open a command prompt window.
- ___ 7. Change directory to **C:\LabFiles\biblio**.
- ___ 8. Run this command to start your application.

```
node index.js
```

- ___ 9. Open a web browser.

__10. Enter the URL: **http://localhost:3000/books**

Biblio Database

ISBN	Title	Price
123	Dune	18

__11. Make sure that the page looks like above. You may get an error if you used tabs and spaces in the book_list.jade file, remember you cannot use both.

Right now there is only one book in the database. Let's add another one.

__12. Open a new command prompt window and change directory to **C:\LabFiles**.

__13. Enter this command to insert a second book. The command is broken up in two lines. But should be entered as one line.

```
curl -v -X PUT -H "Content-Type: application/json"
  -d @book2.json http://localhost:3000/books/456
```

__14. Refresh the browser. Now you should see two books.

Biblio Database

ISBN	Title	Price
123	Dune	18
456	The Stranger on the Train	4.48

Part 5 - Applying Styles

We will now apply CSS styles to various elements.

__1. In book_list.jade file add a style for the header row as shown in bold face below.

```
table
  tr(style="background:black;color:white")
    td ISBN
    td Title
    td Price
```

__2. Save changes.

__3. Refresh the browser and verify that the changes have taken effect.

ISBN	Title	Price
123	Dune	18
456	The Stranger on the Train	4.48

Now we will set the background color of the odd and even rows differently.

__4. First, obtain the index of each iteration by adding the code shown in boldface below (remember adding the comma).

```
each book, index in book_collection
```

__5. Then set the style of the book rows conditionally as shown in bold face below (in one line).

```
each book, index in book_collection
  tr(style= index % 2 == 0 ?
    "background:white" : "background:#cfcfcf")
    td #{book.isbn}
    td #{book.title}
    td #{book.price}
```

__6. Save changes.

__7. Refresh the browser and verify that the changes have taken effect.

Biblio Database

ISBN	Title	Price
123	Dune	18
456	The Stranger on the Train	4.48

Part 6 - Clean Up

__1. End you web service application by hitting **Control+C**.

__2. End MongoDB by hitting **Control+C**.

__3. Close all.

Part 7 - Review

In this lab we learned how to use the Jade template engine to generate dynamic HTML.

Lab 16 - Clustering a Node.js Application

A Node.js application code runs in a single thread. Which means only one of the many CPU cores will be used by this code. To take advantage of multiple cores you will need to run multiple Node.js processes. This is called clustering. In this lab we will learn how to setup clustering using the **cluster** core module.

Part 1 - Getting Started

To save time a simple web application is given to you. Your goal in this lab will be to create a cluster for it.

- ___ 1. Using an editor open **C:\LabFiles\clubby\clubby.js**.
- ___ 2. Note the following aspect of the web application.
 - It listens on port 8000.
 - It returns a simple "Hello World" response.
 - If the URL path is "/throw" then it throws an error. This will terminate the application. We have this code to simulate some kind of unexpected error in the application.

We will now try out the application.

- ___ 3. From a command prompt window go to **C:\LabFiles\clubby**.

```
cd C:\LabFiles\clubby
```

- ___ 4. Run the application.

```
node clubby.js
```

- ___ 5. From another command prompt go to **C:\LabFiles**.

```
cd C:\LabFiles
```

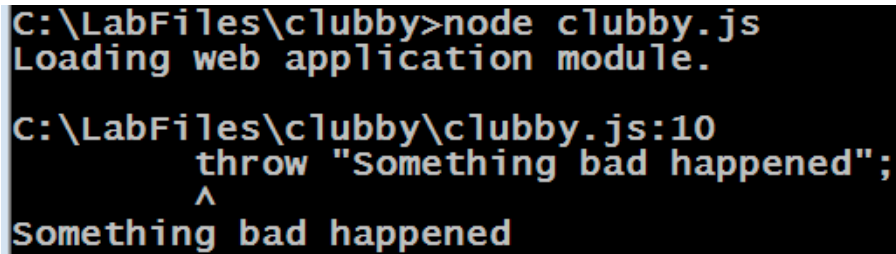
- ___ 6. Send a HTTP request to the application like this and make sure that you see the "Hello World" response.

```
curl localhost:8000/hello
```

__7. Next, send the following request.

```
curl localhost:8000/throw
```

__8. The Node.js application should crash out like shown above.



```
C:\LabFiles\clubby>node clubby.js
Loading web application module.

C:\LabFiles\clubby\clubby.js:10
    throw "Something bad happened";
    ^
Something bad happened
```

Part 2 - Setup a Cluster

We will now run the clubby application in a cluster.

__1. In **C:\LabFiles\clubby** folder create a new file called **index.js**.

__2. Add the following code to **index.js**.

```
var cluster = require('cluster');
var numCPUs = require('os').cpus().length * 2;

if (cluster.isMaster) {
  for (var i = 0; i < numCPUs; i++) {
    console.log("Forking a new child...");
    cluster.fork();
  }
} else {
  //Start the web application
  var clubby = require('./clubby');
}
```

Here, we are spawning a few child processes (twice the number of CPU cores). From a child process we load the clubby module which basically starts the web application.

The exact relationship between the number of CPU cores and the number of child processes you need to spawn depends on the nature of your application. If your application performs very little waiting from various callback functions then spawn as many processes as there are CPU cores. If there is some wasted CPU cycles due to waits then spawn more child processes (as we do here).

__3. Save changes.

To make sure that clustering is working we will log the process ID of the child that handles a request.

__4. In clubby.js add the line shown in bold face below.

```
http.createServer(function (req, res) {  
    console.log("Request served by process: %d", process.pid);  
  
    res.writeHead(200, {'Content-Type': 'text/plain'});
```

__5. Save changes.

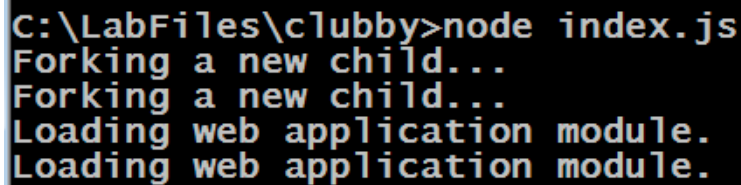
Part 3 - Test

__1. From a command prompt window go to **C:\LabFiles\clubby**.

```
cd C:\LabFiles\clubby
```

__2. Run the application. But this time use index.js.

```
node index.js
```



```
C:\LabFiles\clubby>node index.js  
Forking a new child...  
Forking a new child...  
Loading web application module.  
Loading web application module.
```

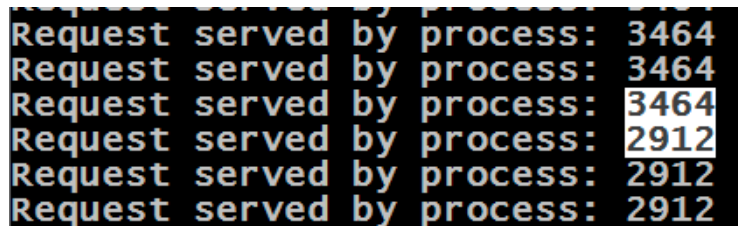
Depending on the number of CPU cores in your machine a few child processes will be spawned.

Now we will send many concurrent requests to the application. We will use the Apache Benchmark tool for this.

__3. From another command prompt go to **C:\LabFiles**.

__4. Run the Apache Benchmark tool like this.

```
ab -c 4 -n 1000 http://localhost:8000/hello
```



```
Request served by process: 3464  
Request served by process: 3464  
Request served by process: 3464  
Request served by process: 2912  
Request served by process: 2912
```

__5. Look at the log from the web application. You should see that different child processes are handling the requests.

Now we will test for error condition.

__6. Send a request as follows.

```
curl http://localhost:8000/throw
```

This will cause one of the child processes to crash. But the other ones will continue to function.

__7. Run the Apache Benchmark tool to verify that the application is still responsive.

__8. Finally, send the "/throw" request a few times until all child processes have crashed. This will cause the parent Node.js process to also end.

Part 4 - Add Failover

We will now write code to spawn a new child process every time one crashes out. The cluster object fires the "exit" event every time a child process ends. We will handle this event and spawn a new child.

__1. In **index.js** add the code shown in bold face below.

```
if (cluster.isMaster) {  
  // Fork workers.  
  for (var i = 0; i < numCPUs; i++) {  
    console.log("Forking a new child...");  
    cluster.fork();  
  }  
  
  cluster.on('exit', function(worker, code, signal) {  
    console.log('Worker process %d has died.', worker.process.pid);  
    cluster.fork();  
  });  
} else {
```

__2. Save changes.

Part 5 - Test

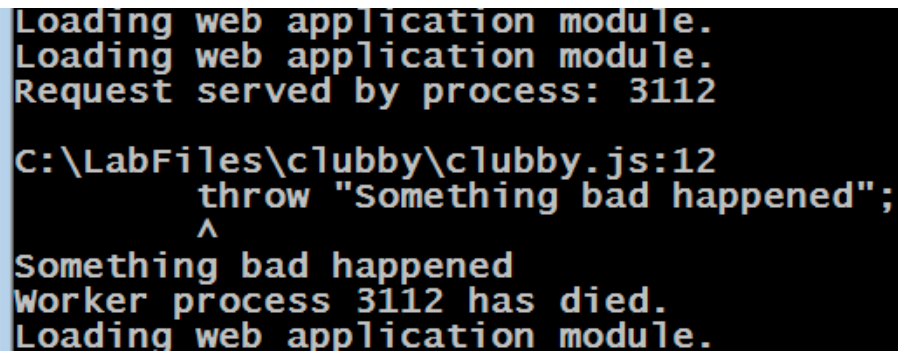
__1. Run the application again.

```
node index.js
```

__2. Send a "/throw" request.

```
curl http://localhost:8000/throw
```

__3. This time a new child process will be launched immediately.

A terminal window with a black background and yellow text. The text shows the application loading a web application module, serving a request from process 3112, and then crashing with an error: 'C:\LabFiles\clubby\clubby.js:12 throw "Something bad happened";'. An arrow points to the error line, followed by the message 'Something bad happened' and 'Worker process 3112 has died.'. The application then restarts by loading the web application module again.

```
Loading web application module.  
Loading web application module.  
Request served by process: 3112  
  
C:\LabFiles\clubby\clubby.js:12  
    throw "Something bad happened";  
    ^  
Something bad happened  
Worker process 3112 has died.  
Loading web application module.
```

Part 6 - Clean Up

__1. End you web service application by hitting **Control+C**.

__2. Close all.

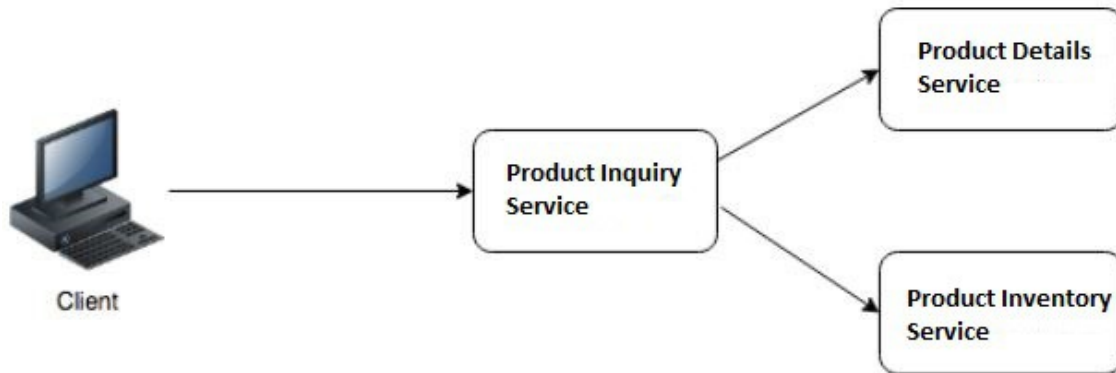
Part 7 - Review

In this lab we learned how to cluster a Node.js application. We also learned how to add failover behavior. In addition to this you should create a Windows service to add another layer of protection. If the parent process crashes for whatever reason Windows will restart it.

Lab 17 - MicroServices with Node

In this lab, we'll experiment with the microservices pattern in Node.js. We will implement a RESTful service that calls two other services to retrieve information that is part of a response.

The setup looks like this:



The Product Details Service and the Product Inventory Service are supplied for you. You will implement the Product Inquiry Service.

The product inquiry service responds to a GET request of the form

```
GET /product/10993
```

The response will be in the form of JSON like so:

```
{
  "productId": 10993,
  "description": "External Monitor",
  "price": 125.24,
  "quantity": 15
}
```

In order to get the data, the Product Inquiry Service sends a call to the Product Info Service:

```
GET /details/10993
```

...receiving the following JSON response:

```
{
  "productId": 10993,
  "description": "External Monitor",
  "price": 125.24
}
```

It also sends a call to the Product Inventory Service:

```
GET /inventory/10993
```

... receiving the following JSON response:

```
{
  "itemId": 10993,
  "quantityOnHand": 15
}
```

In a real microservice environment, we'd use a service discovery mechanism. In this simplified example, however, the locations of the services are coded into a file, 'services.json' (contained in the supporting files), with the following contents:

```
{
  "productDetailService":
    {
      "name"    : "Product Detail Service",
      "server"  : "http://localhost",
      "serviceRoute" : "/details",
      "port"    : "3010"
    },
  "productInventoryService":
    {
      "name"    : "Product Inventory Service",
      "server"  : "http://localhost",
      "serviceRoute" : "/inventory",
      "port"    : "3020"
    },
  "productService":
    {
      "name"    : "Product Inquiry Service",
      "server"  : "http://localhost",
      "serviceRoute" : "/product",
      "port"    : "3030"
    }
}
```

Parts to this Lab

1. Install and test "Product Details" service
2. Install and test "Product Inventory" service
3. Create a basic "Product Inquiry" service
4. Use `http.get` to call the "Product Details" service from within "Product Inquiry" service.
5. Combine promises to return product details and inventory information in the same response.

Part 1 - 1. Install and test "Product Details" service

This lab consists of three services. In this part we will install and test an existing service that returns Product Detail data.

__ 1. In the **C:\LabFiles** folder create a directory called **microservices**.

__ 2. Copy the following files:

package.json

services.json

product-detail.json

From: C:\LabFiles\microservices-files

Into: C:\LabFiles\microservices

__ 3. Also copy the "ProductDetailsService" directory and its files

From: C:\LabFiles\microservices-files

Into: C:\LabFiles\microservices

__ 4. At this point your microservices directory should contain the following:

```
C:\LabFiles\microservices
| package.json
| product-detail.json
| services.json
```

```
└──ProductDetailsService
    data_access.js
    index.js
```

__5. Open the package.json file in your editor. You should see the following:

```
{
  "name": "product-inventory-service",
  "version": "1.0.0",
  "dependencies": {
    "body-parser": "^1.14.0",
    "express": "^4.13.3"
  }
}
```

Node package manager (npm) uses this file to determine which JavaScript libraries to install.

__6. Open a command prompt and navigate to the C:\LabFiles\microservices directory.

__7. Execute the following command:

```
npm install
```

npm will create a "node_modules" directory and install "body-parser" and "express" into it.

If you run into problems executing the npm command install the files instead by unzipping the following file into your microservices directory:

C:\LabFiles\microservices-files\node_modules.zip

__8. The C:\LabFiles\microservices\services.json file is used to provide settings for each of the services in this lab. The section of the file pertaining to the "ProductDetailService" looks like this:

```
"productDetailService":
{
  "name"    : "Product Detail Service",
  "server"  : "http://localhost",
  "serviceRoute" : "/details",
  "port"    : "3010"
}
```

__9. From a command prompt execute the following command from inside the

C:\LabFiles\microservices directory to start the "Product Details Service":

```
node ProductDetailsService
```

You should see the following displayed in the command window:

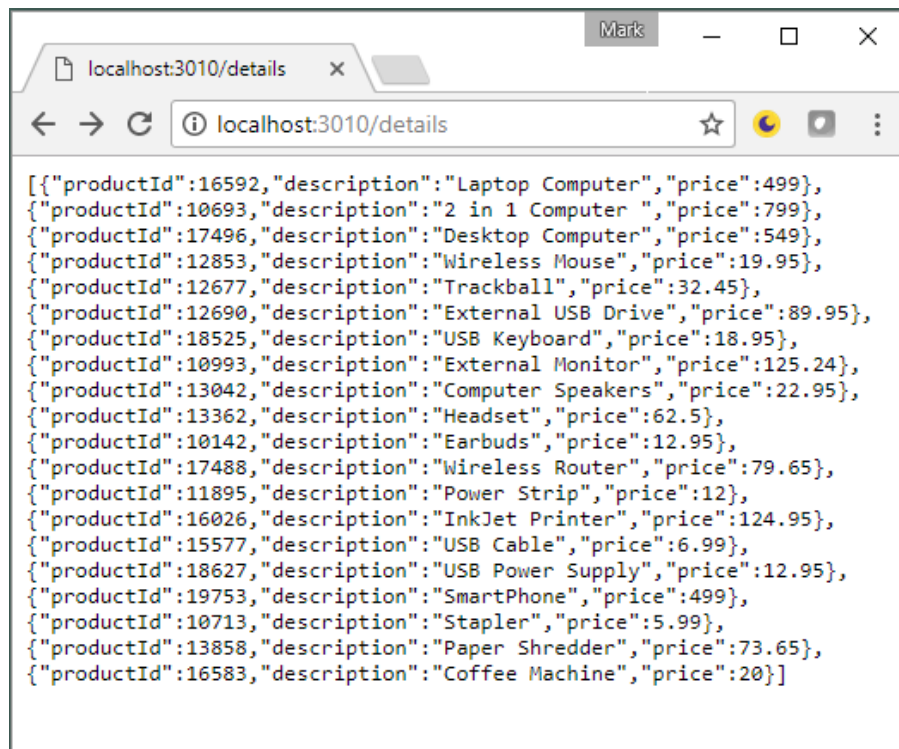
```
Service: 'Product Detail Service'  
Route: /details'  
Port: 3010
```

__10. Open the Chrome browser. We will use it to test the service.

__11. Open the following URL in chrome:

```
http://localhost:3010/details
```

You should see something like this. You want to to adjust the width of the browser window for best viewing:

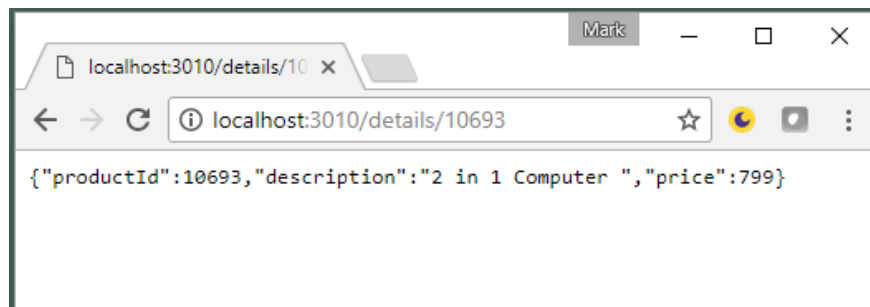


__12. The data that you see here is coming from the C:\LabFiles\microservices\product-detail.json file.

__13. Try retrieving a single record by opening this URL in Chrome:

`http://localhost:3010/details/10693`

Now the browser should show a single record:



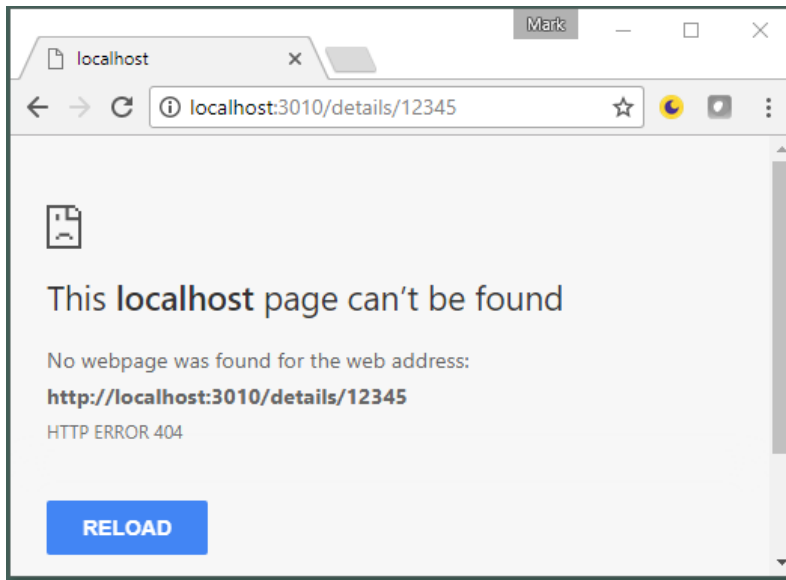
Note that the services we are using/creating in this particular lab are all read-only. This means we will only need to test using HTTP GET requests which can be accomplished quite easily using just the Chrome browser.

That being true, if you are familiar with other tools such as the *curl* command line HTTP utility or the *Postman* Chrome extension feel free to use them instead.

14. Lets see what happens if we try to retrieve a non-existent record. Try opening the following URL in Chrome:

`http://localhost:3010/details/12345`

You should get the following screen:



The message on the screen shows that an HTTP 404 error was returned. The 404 code stands for "not found". The error correctly shows that the record we are looking for "12345" does not exist. You can verify this by checking the `product-detail.json` file.

http://localhost:3010/details/12345
HTTP ERROR 404

Part 2 - Install and test "Product Inventory" service

In this part we will install and test an existing service that returns Product Inventory data.

___1. Copy the following file:

`inventory.json`

From: `C:\LabFiles\microservices-files`

Into: `C:\LabFiles\microservices`

___2. Also copy the "ProductInventoryService" directory and its files

From: `C:\LabFiles\microservices-files`

Into: C:\LabFiles\microservices

__3. At this point your microservices directory should contain the following:

```
C:\LabFiles\microservices
├── inventory.json
├── package.json
├── product-detail.json
├── services.json
├── node_modules
│   ├── body-parser
│   └── express
├── ProductDetailsService
│   ├── data_access.js
│   └── index.js
└── ProductInventoryService
    ├── data_access.js
    └── index.js
```

__4. The C:\LabFiles\microservices\services.json file is used to provide settings for each of the services in this lab. The section of the file pertaining to the "ProductInventoryService" looks like this:

```
"productInventoryService":
{
  "name"    : "Product Inventory Service",
  "server"  : "http://localhost",
  "serviceRoute" : "/inventory",
  "port"    : "3020"
}
```

__5. From a command prompt execute the following command from inside the C:\LabFiles\microservices directory to start the "Product Inventory Service":

```
node ProductInventoryService
```

You should see the following displayed in the command window:

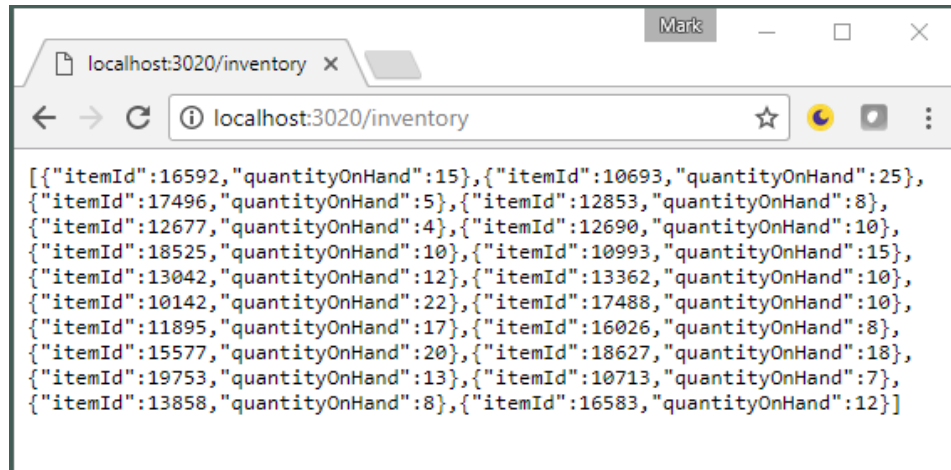
```
Service: 'Product Inventory Service'
Route: /inventory'
Port: 3020
```

__6. Open the Chrome browser. We will use it to test the service.

__7. Open the following URL in chrome:

`http://localhost:3020/inventory`

You should see something like this. You want to to adjust the width of the browser window for best viewing:



__8. The data that you see here is coming from the
C:\LabFiles\microservices\inventory.json file.

__9. Try retrieving a single record by opening this URL in Chrome:

`http://localhost:3020/inventory/10693`

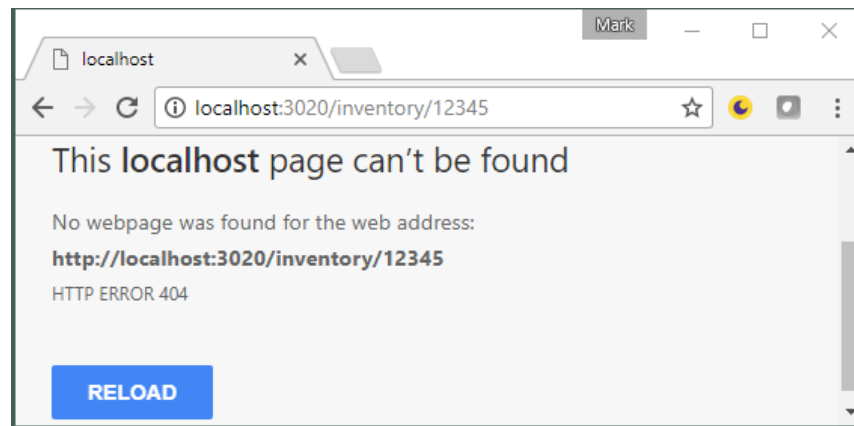
Now the browser should show a single inventory record:



__10. Lets see what happens if we try to retrieve a non-existent record. Try opening the following URL in Chrome:

`http://localhost:3020/inventory/12345`

Just like with the Details service you should get a screen displaying a 404 - not found error. The error correctly shows that the record we are looking for "12345" does not exist in the inventory.json file.



Now that the Details and the Inventory microservices are working we can move on to creating a front-end service to call them and combine their data.

Part 3 - Create a basic "Product Inquiry" service

In this part we will be creating the Product Inquiry service. The first iteration of the service will return a hard coded message. In the parts that follow we will then set it up to retrieve data from the Detail and Inventory services.

- __1. Create a directory named "ProductInquiryService" inside C:\LabFiles\microservices
- __2. Create a file name "index.js" inside the C:\LabFiles\microservices\ProductInquiryService directory.
- __3. Open the "index.js" file you just created in your programming editor
- __4. Add the following contents and then save the file. (if needed this text can be cut and pasted from the following file: C:\LabFiles\microservices-files\pinq-service-001.txt) :

```
var express = require('express');
var bodyParser = require('body-parser');
var fs = require('fs');

var app = express();
var services = JSON.parse(fs.readFileSync('services.json',
'utf8'));
```

```

var name = services.productService.name;
var port = services.productService.port;
var serviceRoute = services.productService.serviceRoute;

app.get(serviceRoute + "/:id", function(req, res) {

    var data = { "message": "Hello from " + name + "!" }
    res.send(data);

});

console.log("Service: '" + name
            + "'\nRoute: " + serviceRoute
            + "'\nPort: " + port);

app.listen(port);

```

This file is set up in a similar fashion to the other two services. It is configured with the following data from the services.json file:

```

"productService":
{
  "name"      : "Product Inquiry Service",
  "server"    : "http://localhost",
  "serviceRoute" : "/product",
  "port"      : "3030"
}

```

As it is set up here the route requires a record id. We will use the id later to retrieve records from the other two services. For now though we just return a hard coded message:

```

app.get(serviceRoute + "/:id", function(req, res) {

    var data = { "message": "Hello from " + name + "!" }
    res.send(data);

});

```

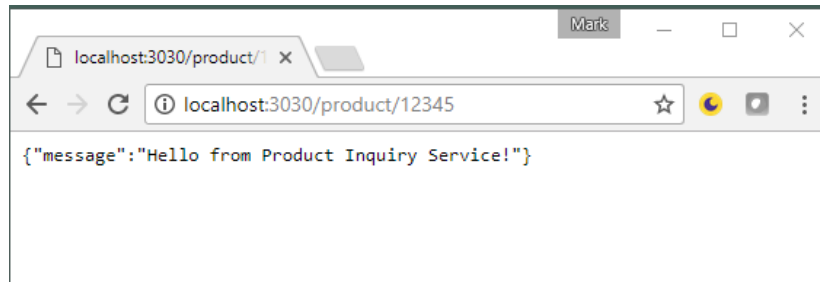
5. Try running the service. Open a new command prompt and navigate to the C:\LabFiles\microservices directory. Then execute the following command:

```
node ProductInquiryService
```

6. Open a new browser window with the following URL:

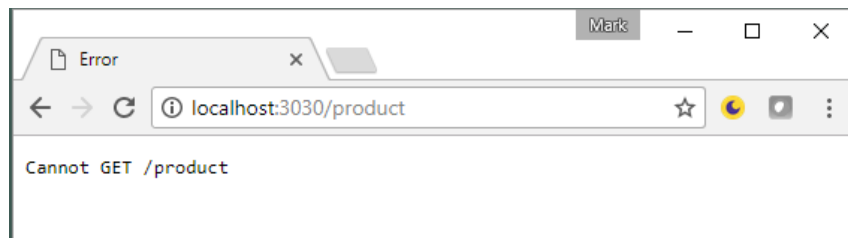
`http://localhost:3030/product/12345`

The browser shows the message we hardcoded into the service. Notice that the product id is required on the URL even though we are not yet using it.



7. Note what happens if you forget to add the product id to the URL. Try loading this URL in the browser:

`http://localhost:3030/product`



The message you get now, "Cannot GET /product" is returned by Express to say that the service, as currently written, does not support HTTP GET requests against the "/product" path when it does not include the id.

In the next section we will build-out the ProductInquiryService so that it calls and returns data from the ProductDetails service.

Part 4 - Use `http.get` to call the "Product Details" service from within "Product Inquiry" service

In this part we are going to use the HTTP package and the promise object inside the Product Inquiry Service in order to retrieve product detail data. We will extend this in the next section to include data from the Inventory service.

1. Create a new file named "`network_api_access.js`" in the `C:\LabFiles\microservices\ProductInquiryService` directory.

2. Add the following code and save the file. (If needed this text can be cut and pasted from the following file: C:\LabFiles\microservices-files\pinq-service-002.txt) :

```
var http = require('http');
var fs = require('fs');
var services = JSON.parse(fs.readFileSync('services.json', 'utf8'));
var details = services.productDetailService;

module.exports.getProduct = function(itemId) {
    const url_details = details.server + ":" + details.port +
    details.serviceRoute + "/" + itemId;
    return callApi(url_details);
};

const callApi = function(url) {
    return new Promise((resolve, reject) => {
        const request = http.get(url, (response) => {
            if (response.statusCode < 200 || response.statusCode > 299) {
                reject(new Error('Load Failed, status code: ' +
                response.statusCode + ", url: " + url));
            }
            const body = [];
            response.on('data', (chunk) => body.push(chunk));
            response.on('end', () => {
                if (body.length > 0) {
                    resolve(body.join(''));
                } else {
                    resolve(null);
                }
            })
        });
    });
    request.on('error', (err) => {
        reject(err);
    })
};
```

Take a look at this code and see what it does.

- It defines a getProduct method
- getProduct calls the callApi method
- the callApi method executes http.get and returns a promise
- callApi also checks for errors in returned content before returning the promise

Handling the errors here will help to simplify the code that calls it.

Next we will need to update the main service file (index.js) in order to use this code.

__3. Open C:\LabFiles\microservices\ProductInquiryService\index.js in your programming editor.

__4. Add the following after the "var fs = ..." line at the top of the file.

```
var net = require("./network_api_access");
```

__5. Replace the contents of the app.get method so that it appears like this. (If needed this text can be cut and pasted from the following file: C:\LabFiles\microservices-files\pinq-service-003.txt)

```
app.get(serviceRoute + "/:id", function(req, res) {  
  
  var promise = net.getProduct(req.params.id);  
  promise.then((data) => {  
    res.send(data);  
  })  
  .catch((err) => {  
    res.statusCode = 404;  
    res.end();  
  })  
};  
});
```

Notice how the code calls the getProduct method. The getProduct method returns a Promise object. On success the data is sent to the caller using "res.send(data)". On error the status code is set and "res.end()" causes it to be sent to the client.

__6. Go to the command prompt where the Product Inquiry Service is running and use Ctrl-C to stop it. You may need to enter Ctrl-C more than once.

__7. Start the service again by executing the following command:

```
node ProductInquiryService
```

__8. Using the Chrome browser try to access the ProductInquiryService with the following URL:

```
http://localhost:3030/product/12345
```

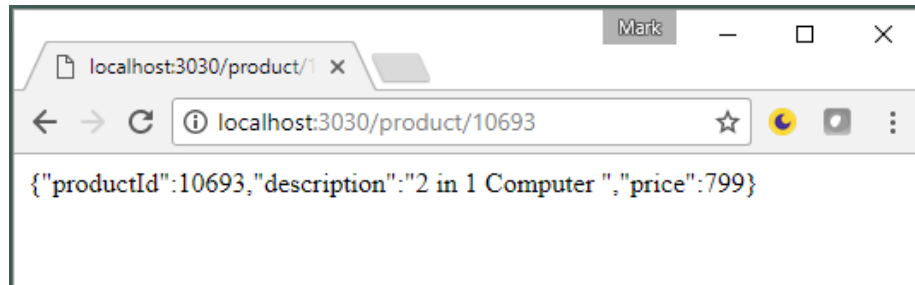
You should get a 404 error this time. The reason for the error is that the id "12345" does not exist in the data that is available from the ProductDetailService that we are now calling.

__9. Try calling the ProductInquiryService with an id that does exist. The following URL

with and id of "10693" should work:

```
http://localhost:3030/product/10693
```

You should see the following:



At this point we have successfully set up the ProductInquiryService to retrieve data from the ProductDetailService. What we really want though is to return data from both of the microservices. We will do that in the next section.

Part 5 - Combine promises to return product details and inventory information in the same response.

In this section we will call both the ProductDetailService and the ProductInventoryService, combine the data and return it from the ProductInquiryService.

__1. Open the C:\LabFiles\microservices\ProductInquiryService\network_api_access.js file in your programming editor.

__2. Add the following after the "var detail = ..." line:

```
var inventory = services.productInventoryService;
```

__3. Replace the contents of the "module.exports.getProduct" method so that it appears as follows. (If needed this text can be cut and pasted from the following file: C:\LabFiles\microservices-files\pinq-service-004.txt)

```
module.exports.getProduct = function (itemId) {  
  
    const url_details = details.server + ":" + details.port  
                        + details.serviceRoute + "/" + itemId;  
    const url_inventory = inventory.server + ":" + inventory.port  
                        + inventory.serviceRoute + "/" + itemId;  
  
    var promise_details = callApi(url_details);  
    var promise_inventory = callApi(url_inventory);
```

```

return new Promise((resolve, reject) => {
  Promise.all([promise_details, promise_inventory])
    .then((values) => {
      if (values != null && values.length === 2
          && values[0] != null && values[1] != null) {
        var obj1 = JSON.parse(values[0]);
        var obj2 = JSON.parse(values[1]);
        if (obj1.productId === obj2.itemId) {
          var result = Object.assign(obj1,
                                     { 'quantity': obj2.quantityOnHand })
          resolve(result);
        } else {
          reject("mismatched results returned")
        }
      } else {
        reject("partial or no results returned")
      }
    })
    .catch((err) => {
      if (err) { console.log(err) }
      reject(err)
    });
});
}

```

The first few lines of this message define URLs and use the callApi() method to return promise objects.

The code that follows has several objectives. It has to:

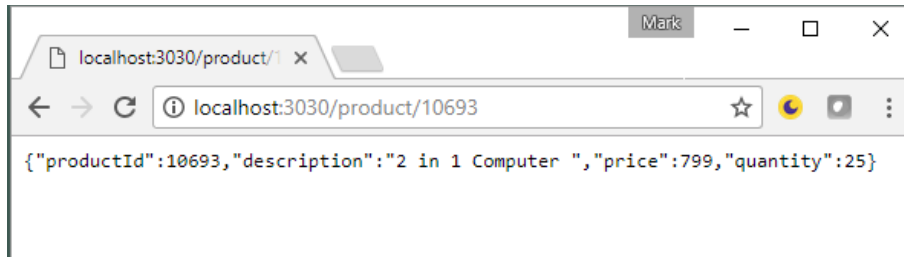
- Combine the promises
- Combine the data when it is returned
- Wrap this all in another Promise that can be passed to the caller.

No changes need to be made to the code in index.js that calls this version of getProduct because it returns a Promise just like it did before.

At this point our ProductInquiryService should be complete. Lets check it out.

___4. Try calling the ProductInquiryService with an id that exist. The following URL with and id of "10693" should work:

`http://localhost:3030/product/10693`



This time the ProductInquiryService returns an object that is a composite of the ones returned by the two services it is calling. The composite object includes "productId", "description" and "price" from the Details service as well as "quantity" which was taken from the Inventory service's "quantityOnHand" property:

```
{ "productId":10693,
  "description":"2 in 1 Computer ",
  "price":799,
  "quantity":25}
```

Many types of services can be created using the same techniques to call multiple back-end APIs and combine their data.

Part 6 - Review

In this lab we created a web service that gets its data from two other back-end services. It uses the HTTP package to access the back-end services and uses the Promise object's "all" method to combine the calls and their results. In this part we will install

Part 7 - Extra credit. Starting services with NPM scripts.

In this lab we've been starting up the services in separate command prompts. In cases like this when you are developing multiple services it can be useful to create npm scripts to start one or more of the services in the same command prompt. The "concurrently" package was created to do just this. In this part we will install "concurrently" and create some scripts that we can use to start up our services.

- __1. Open a command prompt and navigate to the C:\LabFiles\microservices directory.
- __2. Execute the following command to install the Concurrently package. Notice that we are installing it as a Development dependency:

```
npm install --save-dev concurrently
```

___3. After the install completes open and take a look at the package.json file. It should now have a "devDependencies" section like this:

```
{
  "name": "product-inventory-service",
  "version": "1.0.0",
  "dependencies": {
    "body-parser": "^1.14.0",
    "express": "^4.13.3"
  },
  "devDependencies": {
    "concurrently": "^3.4.0"
  }
}
```

___4. Edit the package.json file and add the following "script" section as shown below and save the file:

```
{
  "name": "product-inventory-service",
  "version": "1.0.0",
  "dependencies": {
    "body-parser": "^1.14.0",
    "express": "^4.13.3"
  },
  "devDependencies": {
    "concurrently": "^3.4.0"
  },
  "scripts": {
    "start": "node ProductInquiryService",
    "backend": "concurrently \"node ProductInventoryService\" \"node ProductDetailsService\"",
    "all": "concurrently \"node ProductInventoryService\" \"node ProductDetailsService\" \"node ProductInquiryService\""
  }
}
```

The "backend" and "all" scripts are fairly long and are show word-wrapped above. Make sure that the contents of each command is on a single line in your package.json. (i.e. no carriage returns within the command string)

___5. Before you continue make sure to close out any existing command prompts.

___6. Open a new command prompt and navigate to the C:\LabFiles\microservices directory.

__7. Enter the following command to start up the two background services:

```
npm run backend
```

The command prompt should show something similar to this:

```
[1] Service: 'Product Detail Service'  
[1] Route: /details'  
[1] Port: 3010  
[0] Service: 'Product Inventory Service'  
[0] Route: /inventory'  
[0] Port: 3020
```

The numbers at the front of each line (i.e. [0]) indicate which of the two processes the line's output came from.

__8. Open a second command prompt. Navigate to the C:\LabFiles\microservices directory and enter the following command to startup the ProductInquiryService:

```
npm run start
```

The command prompt should show something similar to this:

```
Service: 'Product Inquiry Service'  
Route: /product'  
Port: 3030
```

At this point all three services are up and running and available for use.

__9. Close the services in each command prompt using Ctrl-C, then close the command prompts.

Next we will use the "all" script to run all three services in one command prompt.

__10. Open a new command prompt and navigate to the C:\LabFiles\microservices directory.

__11. Enter the following command to start up the two background services:

```
npm run all
```

The command prompt should show something similar to this:

```
[2] Service: 'Product Inquiry Service'
[2] Route: /product'
[2] Port: 3030
[0] Service: 'Product Inventory Service'
[0] Route: /inventory'
[0] Port: 3020
[1] Service: 'Product Detail Service'
[1] Route: /details'
[1] Port: 3010
```

All the services are running and available.

In this part we used the Concurrently package to run multiple services from a single command prompt. This technique can be useful during development projects that make use of multiple services.

Lab 18 - Test RESTful API with Supertest

In this lab you will utilize Mocha, Chai, and Supertest for testing RESTful API.

The lab takes a simple Book service which displays existing books and lets you update the books. You will utilize Mocha (test runner), Chai (assertion framework), and Supertest for testing the RESTful API.

Part 1 - Explore an existing service

In this part you will explore an existing service. You will test it using Supertest in later parts of this lab.

__1. Open **Command Prompt** from the **Start Menu**.

__2. Switch to the **Workspace** directory

```
cd c:\software\supertest_service
```

Note: If the folder doesn't exist then extract supertest_service.zip file

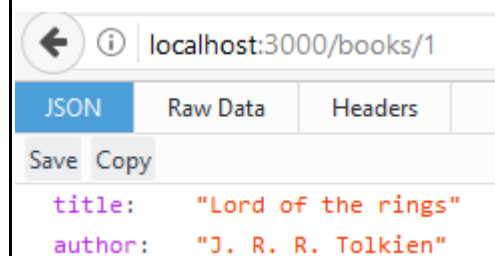
__3. Start the application

```
node index.js
```

__4. Open web browser and enter following URL

<http://localhost:3000/books/1>

Notice the result looks like this



__5. Change URL to

<http://localhost:3000/books/2>

Notice the result looks like this



__ 6. Keep the service running and leave the command prompt open.

Part 2 - Create Directory Structure

In this part you will create directory structure for storing event emitter application

__ 1. Open a new **Command Prompt** window from the **Start Menu**.

__ 2. Switch to the **Workspace** directory

```
cd c:\workspace
```

Note: If the directory doesn't exist, create it by using md [c:\workspace](#) and then switch to it by executing the above command.

__ 3. Create a directory named "events" by using following command

```
md super_test
```

__ 4. Switch to the newly created directory

```
cd super_test
```

Part 3 - Set up a node application

In this part you will initialize a node application and download various modules for testing the RESTful API you explored earlier in the lab.

__ 1. Run following command

```
npm init
```

__ 2. Use default value for each option by pressing the Enter key on the keyboard.

__ 3. Install Mocha

```
npm install -g mocha --save-dev
```

__4. Install Chai

```
npm install chai --save-dev
```

__5. Install Supertest

```
npm install supertest --save-dev
```

__6. Install Express

```
npm install express --save-dev
```

__7. Install Body-Parser

```
npm install body-parser --save-dev
```

__8. Ensure all modules are installed and install them if they are missing

```
npm install
```

__9. Verify modules are installed

```
dir node_modules
```

__10. Create a directory for storing tests

```
md test
```

Note: The directory must be named test. Mocha looks for tests under this directory.

Part 4 - Create tests

In this part you will create various tests for testing the RESTful API which you explored earlier in the lab

__1. Using a text editor, create a new file named **book_test.js** under **test** directory

Note: Ensure that the file is saved under the test directory, otherwise Mocha won't be able to find the tests

__2. Enter following code in book_test.js

```
var expect = require('chai').expect;
var supertest = require('supertest');
var api = supertest('http://localhost:3000');
```

Note: These lines include chai and supertest modules. You are also specifying your RESTful API URL.

__3. Below the above lines, create a test suite

```
describe('Book', function() {
});
```

__4. To the test suite add following test code

```
it('Status code - test', function(done) {
  api.get('/books/1')
    .set('Accept', 'application/json')
    .expect(200, done);
});
```

It's a simple test which makes a service call by using the <http://localhost:3000/books/1> URL. It sets content type to application/json and expects the status code to be 200.

__5. Save the file.

__6. In the Command Prompt window, where you created the test directory and installed node modules, run following command

```
mocha
```

Ensure that you are located under C:\workspace\supertest directory before you execute the above command.

Notice it displays the message that Status code – test has passed

__7. In a text editor, add following code below the above test

```
it('Values - test', function(done) {
  api.get('/books/1')
    .set('Accept', 'application/json')
    .expect(200)
    .end(function(err, res) {
      expect(res.body).to.have.property('title');
      expect(res.body.title).to.equal('Lord of the rings');
    });
});
```



```

        done();
    })
});

```

These lines get the first book by using <http://localhost:3000/books/1> URL, ensures status code is 200, and checks the book title. You can choose to check author in similar way by adding another expect statement to the test.

__ 8. Save the file

__ 9. In the Command Prompt window run following command

```

mocha

```

Notice it displays the message that Values – test has passed

__ 10. In a text editor, add following code below the above test

```

it('Update - test', function(done) {
    api.put('/books/2')
        .set('Accept', 'application/json')
        .send({
            title: 'new book',
            author: 'me'
        })
        .expect(200)
        .end(function(err, res) {

        });

    api.get('/books/2')
        .set('Accept', 'application/json')
        .expect(200)
        .end(function(err, res) {
            expect(res.body.title).to.equal('new book');
            expect(res.body.author).to.equal('me');
            done();
        });
});

```

These lines update the second book contents and checks the contents have been updated.

__ 11. Save the file

__ 12. In the Command Prompt window run following command

mocha

Notice it displays the message that Update – test has passed

__13. In a text editor, add following code below the above test

```
it('Invalid book - test', function(done) {
  api.get('/books/3')
    .set('Accept', 'application/json')
    .expect(404)
    .end(function(err, res) {
      if(err) return done(err);
      done();
    });
});
```

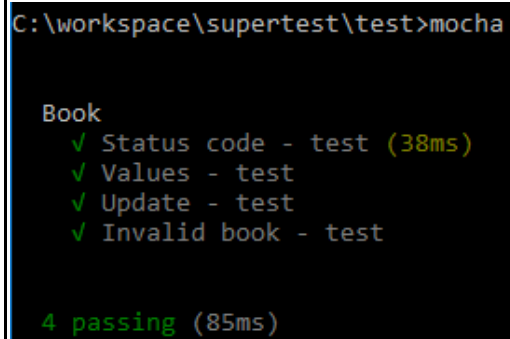
These lines try to access an invalid book, isbn 3, which doesn't exist. It expects status code 404 (not found).

__14. Save the file

__15. In the Command Prompt window run following command

mocha

Notice it displays the message that Invalid book– test has passed. The final output should look like this



```
C:\workspace\supertest\test>mocha

Book
  ✓ Status code - test (38ms)
  ✓ Values - test
  ✓ Update - test
  ✓ Invalid book - test

4 passing (85ms)
```

__16. Close all **Command Prompt** windows

Part 5 - Review

In this lab you utilized Mocha, Chai, and Supertest for testing RESTful API.

Lab 19 - Mock RESTful API with Nock

In this lab you will utilize Mocha, Chai, Supertest, and Nock for testing RESTful API. Nock will be used for mocking RESTful API.

The lab takes a simple Player service which returns existing players. You will utilize Mocha (test runner), Chai (assertion framework), Supertest (testing RESTful API), and Nock (mocking RESTful API).

Part 1 - Create Directory Structure

In this part you will create directory structure for storing event emitter application

- ___ 1. Open a new **Command Prompt** window from the **Start Menu**.
- ___ 2. Switch to the **Workspace** directory

```
cd c:\workspace
```

Note: If the directory doesn't exist, create it by using md [c:\workspace](#) and then switch to it by executing the above command.

- ___ 3. Create a directory named "events" by using following command

```
md mock_test
```

- ___ 4. Switch to the newly created directory

```
cd mock_test
```

Part 2 - Set up a node application

In this part you will initialize a node application and download various modules for testing the RESTful API you explored earlier in the lab.

- ___ 1. Run following command

```
npm init
```

- ___ 2. Use default value for each option by pressing the Enter key on the keyboard.

- ___ 3. Install Mocha

```
npm install -g mocha --save-dev
```

__ 4. Install Chai

```
npm install chai --save-dev
```

__ 5. Install Express

```
npm install express --save-dev
```

__ 6. Install Body-Parser

```
npm install body-parser --save-dev
```

__ 7. Install Supertest

```
npm install supertest --save-dev
```

__ 8. Install Nock

```
npm install nock --save-dev
```

__ 9. Ensure all modules are installed and install them if they are missing

```
npm install
```

__ 10. Verify modules are installed

```
dir node_modules
```

__ 11. Create a directory for storing tests

```
md test
```

Note: The directory must be named test. Mocha looks for tests under this directory.

Part 3 - Create tests

In this part you will create various tests for testing the RESTful API which you explored earlier in the lab

__ 1. Using a text editor, create a new file named **player_test.js** under **test** directory

Note: Ensure that the file is saved under the test directory, otherwise Mocha won't be

able to find the tests

__2. Enter following code in `player_test.js`

```
var expect = require('chai').expect;
var supertest = require('supertest');
var nock = require('nock');
var url = 'http://localhost:5000';
var api = supertest(url);
```

Note: These lines include `chai`, `nock`, and `supertest` modules.. You are also specifying your RESTful API URL.

__3. Below the above lines, create a test suite

```
describe('Player', function() {
});
```

__4. To the test suite, add following setup function

```
beforeEach(function() {
});
```

This setup function will be called before each test case is executed.

__5. To the above setup function, add following code

```
nock(url)
  .get('/players/1')
  .reply(200, {id: 1, name: 'Crosby'});
nock(url)
  .get('/players/3')
  .reply(404, 'Not found');
```

These lines are mocking RESTful API URLs.

__6. Inside the describe block, below the beforeEach setup function, add following code

```
it('Status code - test', function(done) {
  api.get('/players/1')
    .expect(200)
    .end(function(err, res) {
      done();
    });
});
```

__7. These lines are just checking the status code of the API call.

__8. Save the file.

__9. In the Command Prompt window, where you created the test directory and installed node modules, run following command

mocha

Ensure that you are located under C:\workspace\node_test directory before you execute the above command.

Notice it displays the message that Status code – test has passed

__10. In a text editor, add following code below the above test

```
it('Values - test', function(done) {  
  api.get('/players/1')  
    .expect(200)  
    .end(function(err, res) {  
      expect(res.body.id).to.equal(1);  
      expect(res.body.name).to.equal('Crosby');  
      done();  
    })  
});
```

These lines get the first player by using <http://localhost:5000/players/1> URL, ensures status code is 200, and checks the player id and name

__11. Save the file

__12. In the Command Prompt window run following command

mocha

Notice it displays the message the Values – test has passed

__13. In a text editor, add following code below the above test

```
it('Invalid player - test', function(done) {  
  api.get('/players/3')  
    .set('Accept', 'application/json')  
    .expect(404)  
    .end(function(err, res) {  
      expect(res.text).to.equal('Not found');  
      done();  
    })  
});
```

```
    });  
  });
```

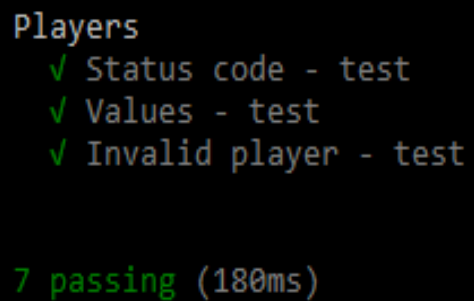
These lines try to access an invalid player which doesn't exist. It expects status code 404 (not found).

__ 14. Save the file

__ 15. In the Command Prompt window run following command

```
mocha
```

Notice it displays the message that Invalid player – test has passed. The final output should look like this



```
Players  
  ✓ Status code - test  
  ✓ Values - test  
  ✓ Invalid player - test  
  
7 passing (180ms)
```

Part 4 - Review

In this lab you utilized Mocha, Chai, Supertest, and Nock for testing RESTful API.

