Node.js Lab Notebook

This contains all the instructions for all the labs in this course. Feel free to print the notebook before class.

Table of Contents

Asynchronous Programming Techniques

In this lab, you will familiarize yourself with various asynchronous programming techniques, a mainstay of Node development.

Objectives

In this lab, you will learn to

- manage scope when using the `this` keyword in the presence of asynchronous callbacks, and
- see how to break up long-running tasks in order to improve Node's performance.

Managing Scope in Callbacks

First off, let's have a look at how you might write a class to encapsulate a filename and a method to determine whether or not that file exists to see how the use of this might be problematic.

Using `this` in callbacks

1. Open file `scope-this.js` in your IDE or text editor. Notice that we're using the `function` keyword to define a class called `FileThing` with a `name` property and a method called `exists` that asynchronously determines whether the file referred to by `this.name` exists.

In particular, have a look at how we're calling `fs.open` then inspecting the error given to us in the supplied `onOpened` callback. If we couldn't open the file, we ensure that it was because it didn't exist, log that fact, then inform our caller appropriately. Otherwise, we successfully opened the file, so we close it via `fs.close` (skipping a close callback) and invoke the supplied callback with no error and a `true` flag.

Observing the loss of `this`

2. Now, use `node` to execute `scope-this.js`. You should see output similar to the following (highlighting added for emphasis).

```
$ node scope-this.js

Opening: thisFileDoesNotExist

/lesson-30-asynchronous-programming-techniques/lab/scope-this.js:13

        console.log('Failed opening file: ' + this.name);
```

```
                                                    ^

TypeError: Cannot read property 'name' of null

      at onOpened (/lesson-30-asynchronous-programming-techniques/lab/scope-this.js:13:53)

      at Object.oncomplete (fs.js:107:15)


Process finished with exit code 8
```

You should see that the file indeed does not exist, but notice the second line of output, where we log that the file wasn't found. It says "Failed opening file: undefined". This meant that the expression `this.name` returned the JavaScript literal `undefined`! Why?

The scope of `this`, at this location, is the enclosing function, `onOpened`. Since it doesn't define a property named `name`, JavaScript correctly returns `undefined`! Let's fix this (pun intended).

Set a reference to the `this` we meant

3. Create a new file called `scope-self.js` and copy the contents of `scope-this.js` into it.

(i) *Remember that the scope of `this` in functions defined in a class is the class's members.*

4. Since the `this` we meant in the `onOpened` function was `FileThing`'s `this`, which is present in the function `exists`, create a new variable inside function `exists` called `self` and set it to the `this` that we meant (that was a mouthful).

5. Replace all uses of `this` in `exists` and in the `onOpened` callback with `self`.

6. Execute `scope-self.js` via `node`, and you should the following output, which properly echoes the filename in the "Failed opening file" message.

```
$ node scope-self.js
```

```
Opening: thisFileDoesNotExist
```

```
Failed opening file: thisFileDoesNotExist
```

```
file does NOT exist
```

When you see the output, you may move on to the next step.

Breaking up long-running tasks

Remember that Node is single-threaded, meaning that it is possible for our code to cause Node to halt all work. This is obviously undesirable, so let's have a look at how we can refactor code to yield its processing, letting Node catch up on I/O that may have been queued while Node was processing our logic.

> *Note that the code for this lab is already completely written and serves instead as an example of yielding.*

This example presents a brute-force means of calculating prime numbers up to a given exclusive limit, in order to provide a good example of long-running, compute-intensive code.

7. Open file `prime-obnoxious.js`, which contains the function `computePrimes`. Notice that the function takes a `limit` greater than or equal to one, handles some boundary conditions, then jumps into the main loop of calculating prime numbers. The loop starts with prime candidate 3 & factor 2 and increments each candidate once it's found to have an integral factor other than itself or all factors have been exhausted.

8. Run file `prime-obnoxious.js` via `node`. You should see output similar to the following. Your timing may vary depending on your hardware and current load.

```
$ node prime-obnoxious.js

9593 primes

3.033 secs
```

> ⚠️ *During the period that function `computePrimes` is running, Node can do nothing else. Other I/O-bound tasks, if present, would be queuing up and waiting for Node's event loop to finish executing `computePrimes` before they could be handled.*

9. Now open file `prime-polite.js`. The first thing that you should notice is that we've introduced into `computePrimes` an inner function `computePrimeBatch`, also known as a JavaScript *closure*, that breaks up the task of computing primes into smaller chunks.

A JavaScript closure is roughly a function defined within another function that captures (or "closes around") and remembers all independent variables that are within scope at the time the closure is executed.

After each chunk has evaluated a certain number of prime candidates, it yields control to Node via the `setImmediate` function so that Node can allow other tasks to execute.

There are two ways of yielding control in Node: `setImmediate` and `process.nextTick`. `setImmediate` queues the given function behind whatever I/O event callbacks that are already in the event queue, ensuring I/O is not being starved. `process.nextTick`, however, queues the given function at the head of the event queue so that it executes immediately after the current function completes, which may starve I/O. In general, prefer `setImmediate`.

10. Run `prime-polite.js` via `node`. You should see output similar to the following. Again, your timing may vary depending on hardware & load.

```
$ node prime-polite.js

9593 primes

3.681 secs
```

Notice the time it took the polite version of the script to complete was about 20% slower than the obnoxious version. Why? Because we're yielding our algorithm's execution to Node so that Node can process any I/O-bound tasks that may be queued, which takes time.

While the performance of our algorithm suffers, the overall throughput of Node remains high because we're giving Node a chance to service any I/O-bound tasks that may have been waiting during our algorithm's processing.

This lab is now complete!

Modules

In this lab, you will create a Node module that performs simple calculations for various shapes.

Objectives

In this lab, you will learn to

- create a Node module, and
- use the newly created module within a calling application.

We are going to create a directory module named 'shapes', which resides in the lab folder. We must create the 'package.json' file to indicate the name and the JavaScript program. Our module will export three additional modules for rectangle, triangle, and circle.

Set things up

Let's start our setup by creating an appropriate place for the module we're going to create.

Create a shapes directory

1. Create a directory called `shapes` and create a `package.json` file within it. Remember a `package.json` needs keys `name` and `main`. Go ahead and use `index.js` for the `main` key and we will create it next.

Ok, we're now ready to start coding our module!

Create the module

Since we want our shapes module to be flexible enough to handle multiple types of shapes, let's create an index file and reference the individual supported shapes there.

Create an index.js file

2. Create an index.js file in the shapes directory and add `require` statements for `rectangle`, `circle`, and `triangle,` which we are about to create.

Export rectangle, circle, triangle

3. After the require statements, export each of them, making them available to users of our module.

> **(i)** *A module can require the use of other modules internally but not necessarily export them. For example, a popular node module `express` requires `cookie`, a separate cookie parsing module internally.*

Create rectangle.js, circle.js, and triangle.js

4. Now that the index file is complete, let's create implementation files for each shape. In case your geometry is rusty, here are some formulas you will include as functions in each shape.

Circle

- $a = \pi r^2$
- $c = 2\pi r$

Rectangle

- $a = wh$
- $p = 2(h + w)$

Triangle

- $a = \frac{1}{2}bh$

You can export each function you create with the syntax:

```
exports.name = function(…) {…}
```

For objects with multiple functions, an alternative syntax is:

```
module.exports = {

   someFunction: function(…) {…},

   someOtherFunction: function(…) {…}

}
```

Write the main program logic

It's time to write the application that will use our newly created module.

Create geometry.js and require the shapes module

5. Create a geometry.js file and `require` the shapes module you created.

Compute some interesting shape values!

6. All that is left now is to write some code using the functions that you exported from your module.

When you're seeing the messages that you expect, you have completed the lab!

Debugging

In this lab, you will debug a very simple JavaScript program to see one of the issues of debugging asynchronous applications.

Objectives

In this lab, you will learn how essential breakpoints are when debugging in node.js.

Open the file in your IDE and debug

1. Open `debug.js` in your IDE. Set a breakpoint on the first `console.log()` call in the file and then debug the file as a node.js application.

Step through the code

2. Try stepping through the code one statement at a time. When you hit the `setTimeout()` call, you will find yourself deep in node.js code.

Set a breakpoint

3. Kill the debugging session and set a breakpoint in the second `console.log()` call that is inside the callback for the `setTimeout()` call. Restart the debug session and now you can use "continue" to get past all of the node.js internals and into the body of the callback.

Optional: Use the node.js debugging client

4. Open up a terminal and navigate to the folder with the debug.js file in it. Enter the command `node debug debug.js`. The debugger will show the top of the file. Enter `help` at the prompt for a list of available commands.

Testing

In this lab, you will familiarize yourself with the module Mocha, a popular unit test framework for Node.

Objectives

You will learn to use the Test Driven Development (TDD) interface of Mocha by creating three simple unit tests.

Install Mocha

The first thing we have to do is to use Node Package Manager, `npm`, to install the `'mocha'` module. Do this by opening a command prompt in the lesson directory and issuing the command `npm install -g mocha`. The `-g` flag installs the module in the global `node_modules` and provides the `mocha` command line interface.

Lab Steps

Now that Mocha is installed, we can begin the lab, whose goal is to create three unit tests for a simple Product class, which has been provided for you in `product.js`.

Run the empty test suite

An empty test suite has been created for you in `productTest.js`. The Product class is instantiated and the Mocha suite definition is set up although it is currently empty.

1. Run the test suite by issuing the following on the command line: `mocha productTest.js -u tdd`. If everything is working correctly, you should see a message in the console indicating that there are 0 passing tests. This makes perfect sense because we do not yet have any tests created! Mocha defaults to Behavior Driven Development (BDD) syntax but for this lab, we will be using the TDD syntax, which we indicate by using the command line option "`-u tdd`". The main differences in BDD and TDD are the syntax used to describe tests but functionally, the test cases should behave the same.

Create the delete test case

2. Create the first unit test declaration with the signature `test('deleteTest', function deleteTest() {…})`. The first thing the unit test should do is `add` a product using the API. Use Node's `assert` module to make sure the product count is 1 by invoking `assert.equal()`. Then run the `deleteAll` API and again assert that the count is now 0.

3. Execute the unit test with the command line `mocha productTest.js -u tdd` and verify that there is one passing test.

Create the add test case

4. Create another unit test declaration with the signature `test('addTest', function addTest() {…})`. Use the `deleteAll` API to reset the product list and then use the `add` API to create one. Use either `assert.equal` or `assert.notEqual` to test for the expected result and run your test suite again from the command line.

Create the doAsync test case

Mocha allows you to test asynchronous code by adding a callback to the test function (usually named `done`). By adding the callback, Mocha knows that it should wait for completion. There is a 2 second timeout by default for a test case but this is configurable with a command line option: `-t`.

6. Create a unit test with the signature `test('doAsyncTest', function doAsyncTest(done) {…}`. The Product class contains a `doAsync` function which makes use of Node's `setTimeout(callback, delay, args…)` API. After waiting for 1.5 seconds, the value of `true` is returned. The test case should call the API with `product.doAsync(function callback(value) {…}` and assert that value is `true`. Then call the `done()` callback to indicate to Mocha that the test case is complete. If you modify the Product class to increase the timeout to a value greater than 2s, you can see that the test case fails because of Mocha's default timeout value.

Once you have created all three unit test cases and they are passing, you are complete with this lab!

The file `productTest-bdd.js` in the solutions folder is an example of tests done in Behavior-Driven Development (BDD) style.

A convenient mocha switch is "–w": this will cause mocha to monitor the current working directory and rerun tests upon changes. Try it out!

Working with the File System

In this lab, you will learn to use Node's File System module by watching a directory for file updates and file & directory creations and deletions.

Objectives

In this lab, you will learn to

- use various functions from Node's File System module, and
- write asynchronous callback functions to handle events from the file system.

Set things up

Let's start by setting up our script with a little bookkeeping.

Require the File System module

1. Open the JavaScript file `watch.js` and declare a variable for Node's File System module, by adding the line

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

Declare a data structure to keep track of files & directories encountered

2. Declare a data structure that will hold our initial view of files and directories. You can use what you want, but an obvious choice would be map of lists:

```
var entries = { files : [], dirs : [] };
```

Store the directory to watch as a variable

3. Declare a variable to hold the name of the directory that we'll be watching (due to the behavior of the file-related callbacks that we'll be using later on):

```
var dir = './dir';
```

Ok, we're now ready to start coding our directory watching logic!

Initialize the data structure with files & directories

The first thing that we need to do is write logic to populate our data structure with the initial lists of files and directories in the directory that we're watching.

Read the directory

4. Use the function `fs.readdir` to get all of the entries in the directory `dir`.

> (i)   `fs.readdir` *takes the directory you want to read and a callback of the form* `function(err, filenames)`*, where* `filenames` *is an array of filenames relative to the directory being read.*

> (i)   *It is a good idea to always name your callback functions, because when things go wrong, you can identify them more easily in stack traces. Also, remember that on success,* `err` *will be falsey, otherwise truey; if it's truey, make sure to throw it.*

Add file & directory names to the data structure

5. Use `Array`'s `forEach` method to process each entry. `forEach` takes a callback of the form `function(element, index array)`.

> (i)   *The* `fs.readdir` *function, when invoking your callback, returns filenames that are relative to the directory being read. For this particular use case, we'll only need to use the first argument (the array element), so your callback can take the form* `function(filename)`*.*

6. In your `forEach` callback, first prefix the given filename with the name of the directory being read.

7. Use `fs.stat` to find out if the given `filename` represents a file or directory.

> (i)   `fs.stat` *takes a callback of the form* `function(err, stats)`*, where* `stats` *is an object of type* `fs.Stats`*, and has methods* `isDirectory()` *and* `isFile()`*. Use those methods to determine if the given* `filename` *represents a file or directory, and add it to the appropriate list (via* `Array`*'s* `push` *method).*

Run what you've got so far

This is a good place to see whether things are working so far.

8. Sprinkle some `console.log` statements in with your `forEach` callback to see that your data structure is getting populated properly. If your data structure is a map called `entries`, you can just log the map itself with a call to `console.log(entries)`.

9. Run your code either in your IDE or at the command line by invoking `node watch.js` in the directory containing the file `watch.js`.

Once you see that your code is populating your data structure ok, move on to the next step!

Write the main program logic

It's time to write the meaty part of our application.

Watch the target directory

10. Use the `fs.watch` function, which takes the target directory and a callback of the form `function(event, filename)`. Write the call to `fs.watch` and include an empty callback for now; we'll complete the callback next.

⚠️ *`fs.watch` emits the `rename` event for all file & directory deletions, creations & renames. On some operating systems (like Mac OS X), a rename actually consists of a deletion and subsequent creation. Your mileage may vary.*

Complete the directory event callback function

11. Make the first line of your `fs.watch` callback function replace the contents of the `filename` variable with the filename prefixed with the target directory.

ℹ️ *Just as the `fs.readdir` callback receives filenames relative to the directory it was given, so does the `fs.watch` callback.*

The `event` given to your callback is actually a string with one of three values: `'error'`, `'change'`, or `'rename'`.

12. Write skeleton code to handle each event value (hint: this is a good place to use a `switch` statement).

Handle errors

13. Write code to handle the `'error'` event. Simply throw an `Error` object with a message that indicates there was an error watching the target directory.

Handle changes

14. Write code to handle the `'change'` event that logs a message indicating which file changed and what its new size is. Use `filename` and the function `fs.stat` again to get the size of the file and echo the filename and its new size to the console (via `console.log`). Don't forget to add your `break` statement if you're using a `switch`!

15. Run your code again, then open and save a file in the directory you're watching. You should see messages indicating that file's changes.

Once that's working, move on to the next step!

Handle file & directory deletions, creations & renames

16. Write code to handle the `'rename'` event, using `fs.exists` to determine whether or not the given file exists. If it does, then you know that it's a creation.

> (i) `fs.exists` *takes a filename and a callback of the form* `function(exists)`, *where* `exists` *is a* `boolean` *indicating whether the file exists.*

17. If `fs.exists` calls back with a `true` value, use `fs.stat` to determine whether it's a file or directory, and log a message to the console stating which type of entry (file or directory) that `event` is for, the name of the file or directory involved, and finally, how many files & directories are now being watched (information which can be gleaned from your data structure).

18. If `event` is for a file or directory that no longer exists, then you know that it was a deletion.

> (i) *You can tell whether or not the file or directory existed by using your data structure; simply see if the file or directory name is present in your list of files or directories, respectively.*

Remove the entry from the appropriate list (via `Array`'s `slice` method), and then log to the console which type of entry `event` was for (file or directory), the file or directory's name, and the number of files and directories now being watched.

19. Run your application again. This time, trying adding then removing files or directories in the target directory and see what happens.

When you're seeing the messages that you expect, you have completed the lab!

Working with Streams & Buffers

In this lab, you will learn to use Node's Stream & Buffer classes to read, compress & write data.

Objectives

In this lab, you will learn to

- leverage Node's ability to stream & manipulate data via pipes.

Compress a file via streams

In the first part of this lab, we're going to compress a file using streams.

Write the main compression logic

1. Create a file called `file.js` and in it, require the modules `fs` and `zlib`.

Create a file reading stream

2. Use `fs`'s `createReadStream` function to create a `stream.Readable` that reads from our source file, `file.js`. Make sure to hold on to the reference returned.

Create a duplex compression stream

3. Next, create a gzip compression stream via `zlib`'s `createGzip()` function. Note that this is a `Duplex` stream, meaning it is both `Readable` and `Writable`.

Create a file writing stream

4. One more stream that we'll need is a `Writable` stream for our output file. Use `fs`'s `createWriteStream` function to create a writable file stream to a new file, `file.js.gz`.

Pipe from the file to the output stream via the compression stream

5. With all of our streams now set up, we can pipe from the read stream to the compression stream, then on to the write stream. Write a chaining pipe expression that begins with our file reading stream, goes through our compression stream, and finishes with our writable stream and execute `file.js` via `node`.

Confirm your results

6. Confirm your results by checking the newly created gzip file `file.js.gz`. If you have `gzip` in your path, a simple `gzip -t file.js.gz` will suffice; if it returns with no error, then the file is a valid gzip archive, otherwise it'll echo an error.

When you've confirmed that your archive is being created properly, move on to the next step.

Compress URL stream

The next part of this lab is optional and requires Internet access. If you have it, continue on. If not, you're done with this lab!

Add required modules

7. Open file `net.js` and require the following modules: `fs`, `zlib`, & `https`.

Instantiate streams

8. Instantiate two streams: one for compression via `zlib.createGzip()`, and one for writing to a file (where we'll write the response we'll be getting from the URL).

Invoke the URL

9. Use the `https` module's `request` function, whose first argument is an object with properties `hostname`, `path` & `method` (among others – check out the docs if you're curious), and whose second argument is a callback that takes a `response` object. Use `'www.google.com'` for the `hostname`, and `'/?q=node.js'` for the `path`, and `'GET'` for the `method`.

Capture the returned object and make sure to call its `end()` method so that the server knows that the request is complete and it should start responding. Alternatively, you could just call `end()` on the return value of `https.request`, without even storing the variable, since we're not going to use it after the `end()` call.

Chain the response

10. In the event handler, take the response stream, pipe it to the compression stream, and then pipe that to the writable file stream.

Run it & test

11. Now that you've got the code written, run `net.js` via `node`, extract the newly created `google.gz` file, and make sure that it's innards look like what Google would return in a query for node.js.

Once you see the results you expect, you're done with this lab!

Networking with Node

In this lab, you will learn to use Node's low-level networking capabilities to create a simple TCP echo server and interact with it from a network client.

Objectives

In this lab, you will learn to

- use various server & client functions of the `net` module,
- use the `net.Server`, `net.Socket`, and `Buffer` objects, and
- write asynchronous callback functions to handle events from sockets.

Write the server logic

We'll jump right into writing the server.

Create a socket server

1. Create & store in a variable called `server` a new socket server via `net`'s `createServer` factory function. For its callback, use the function `onClientConnected`, whose stub has already been provided for you.

Start listening on a port

2. Tell the server to start listening on port `PORT`, and provide the function `onServerPortBound` as the callback, whose full implementation has also already been provided for you.

Test the server

At this point, believe it or not, we have a functioning TCP server that can listen on port `PORT`, and accept connections & handle disconnections.

3. Execute `server.js` using `node`. You should see a message that looks something like the following.

```
$ node server.js

server listening on port 1986
```

4. Open another terminal window and enter the command `telnet localhost 1986`. You should see output similar to the following.

```
$ telnet localhost 1986

Trying 127.0.0.1...

Connected to localhost.

Escape character is '^]'.
```

Kill the server (via `Ctrl-C` if in a terminal) and you should see the client terminal echo something similar to the following.

```
Connection closed by foreign host.
```

When you are confident that the server is running and you're seeing these messages, move on to the next step.

Start tracking client connections

We're now going to start adding logic to our `onClientConnected` function to keep track of the connections that this server has made.

5. First, push the `socket` given to the callback onto the `sockets` array for later use.

Add a disconnection handler

6. Now, add a listener to the disconnection event of the socket passed into `onClientConnected`; the event is `'end'` and the callback function we'll use is an inline function called `onEnd` that delegates to `onSocketEnd(socket)`.

> *To add a listener to any* *EventEmitter* *(like* *net.Socket*)*, use* *EventEmitter*'s on method, which takes the event of notification and the callback function.

> *The inline function that we created is also a closure, since it uses the socket given to the* *onClientConnected* *callback.*

Test the server

Now that we've coded a little more, let's test a little more.

7. Run `server.js` via `node` again.

8. Once you see the message that the server is listening, connect to it from another terminal again.

9. Now that the client is connected, type `Ctrl-]` (the close-square-bracket key, or whatever your terminal states the escape character is) to escape to the telnet prompt, then type `quit` and enter. The client terminal should look something like the following.

```
$ telnet localhost 1986

Trying 127.0.0.1...

Connected to localhost.

Escape character is '^]'.

^]

telnet> quit

Connection closed.
```

10. Confirm that your client has closed the connection and that you see the server's disconnection message. The server's messages should look something like the following.

```
$ node server.js

server listening on port 1986

client connected

client disconnected
```

Next, let's write some usage information to the client so that the client knows how to interact with the server.

11. Use the following code at the end of the `onClientConnected` function to write the usage message to the client.

```
socket.write('Hello.  Usage:\r\n'
  + '/name:yourName to tell me your name\r\n'
  + '/quit to quit\r\n'
  + '/shutdown to shutdown the server\r\n'
  + "otherwise I'll just echo 'You said: ' plus what you entered!\r\n");
```

Now, let's add logic to handle the behavior that the usage message describes.

Add a data listener

12. Right after the line(s) in `onClientConnected` that register the `'end'` listener, add another listener for the `'data'` event, this time creating an inline function called `onData` that takes a single argument named `data` and that delegates to `onSocketData(data, socket)`, another function whose stub is provided for you.

> (i) The `'data'` event from a socket passes in data as an instance of Node's `Buffer` object, unless the stream's `setEncoding` was previously called with a valid encoding string, in which case data is given as a string with the desired encoding.

We're now going to complete the implementation of the behavior described in the usage that's given to clients right after they connect, namely that any string is echoed back to the client with the prefix `'You said: '`, except for `'/quit'`, which closes the client connection, `'/shutdown'`, which shuts down the server completely, or `'/name:yourName'`, which causes the server to prefix echoed strings with `'<yourName> said: '`. All of these edits will be going into the function `onSocketData`.

13. Start by setting `data` to `data.toString()`, just to make sure it's a string.

14. Add a check to see if the string is literally `'/shutdown'`. If it is, invoke `shutdown()` and return. We'll flesh out the function `shutdown` later.

15. Add a check to see if the string is literally `'/quit'`. If it is, invoke `shutdown(socket)`. Again, we'll flesh out the function `shutdown` later.

16. Next, see if the string begins with `'/name:'`. If it does, then parse the string following the colon, and save it as a new property, `username`, on `socket`.

ⓘ *You can use `String`'s `indexOf` method to see if a string begins with another string.*

ⓘ *Remember that in JavaScript, you can add arbitrary properties to any object. Our requirement is to track the username for each client independently. Since each socket already uniquely identifies a client, we can just add a `username` property to the socket and set its value to the name the user gave.*

17. Write to the socket a message that the server will now call the user by their new name and return.

18. If none of the prior conditions were met, then simply write to the socket `'<name> said: '`, then the string they entered, where `<name>` is the current value of `socket.username`.

19. Lastly, give `socket.username` a default value of `'You'` at the end of the `onClientConnected` function.

Test the server

20. Now that we've coded a little more, let's test a little more. Run the server, connect with a telnet client again, and make sure that messages are being echoed, that `'/name:'`, `'/quit'`, and `'/shutdown'` messages are being handled correctly.

Once your server is behaving properly, move on to the next step.

Implement behavior for /quit and /shutdown messages

21. Now, add logic to the shutdown function to handle the `'/quit'` and `'/shutdown'` messages.

In function `shutdown`, if socket is truey (basically, not `null` or `undefined`), find `socket` in the `sockets` array, call its `end()` method, then remove it from the `sockets` array and return.

> ℹ️ *You can use `Array`'s `indexOf` method to find the index of the element you're looking for. If it's not found, `indexOf` will return `-1`.*

If `socket` is falsey (basically, `null` or `undefined`), then the shutdown function is supposed to interpret the call as a shutdown of the entire server, including all client connections (which is, of course, something that you probably would *not* want to do in a real server application). First, cause the server to stop accepting new connections via `Server`'s `close` method and give it an inline function called `onClosed` that simply logs to the console that the server has been closed. Then, loop through the `sockets` array, telling each socket to `end()`. Make sure to stop referencing each socket object, after you've closed it!

> ⚠️ *Note that `server.close()` does not actually shutdown the server and close all existing client connections. It merely stops the server from accepting new connections; you must still close any open connections the server has.*

Test the server

22. By now, you know the deal: code a little, test a little. Make sure the server now closes the client connection on `'/quit'` messages and shuts down the entire server on `'/shutdown'` messages.

Once your server is behaving properly, move on to the next step.

Handle expected server errors

The last step in this lab will be to add some reasonable error handling in the server.

23. Add a server error listener just before the line where you invoke `server.listen`. The event we want to listen for is `'error'`, and our callback can be written inline with the name `onError` and a single parameter for the Error object.

The only error that we're going to handle in this lab is when the port is already in use by some other process. In that case, the error's `code` property will be `'EADDRINUSE'`. Check for that condition, and if it's met, invoke `server.listen` again after a one second timeout and return. Otherwise, we'll just let any other error crash the server's initialization and log the error to the console.

Test the server

24. Run one instance of the server in order to occupy the port; make sure it successfully starts up.

25. Now, run another instance of the server in another terminal and make sure that the `onError` handler is getting called and issuing a deferred `server.listen` call each second.

26. After a few failed attempts by the second server to start, kill the first server (however you want to). This will free the port, allowing the next invocation of `server.listen` in the second server instance to succeed.

When you see the second server process running successfully, you have completed the lab!

Web Servers

In this lab, you will use most of the key components of the node.js http module.

Objectives

In this lab, you will learn to

- create HTTP servers and make client requests,
- read and write data to and from the HTTP server.

Start your webserver

1. Open `webserver.js` in your IDE. This essentially the same "Hello World" server as we saw in the introduction. Run this file from your IDE or from the command line. Navigate to <u>http://localhost:3000</u> with your browser and you should see "Hello World" in your browser.

Add support for different clients

2. Shut down the server for now. We're going to add the ability to respond differently to different clients using the value of the "Accept" header. Add code to the `requestListener()` method to send "Hello World" back to the client in an H1 HTML element if the client accepts 'text/html.' You have access to all the headers on the `request.headers` property. Restart the server and you should now see an HTML version of "Hello World" in your browser.

You can also try to run `curl http://localhost:3000`, to see the example of plain response.

Use parameters in the query string

Add a supporting module

3. Shut down the server again. Include the `url` module by adding a `requires` statement and assigning it to a variable.

Read the values in the query string

4. Go back to the `requestListener()` method and use the `url.parse()` method to parse the URL that is available on the `request.url` property. Use `true` for the second argument to the `url.parse()` method so that the query string will be available. Assign the return to a variable.

Change the HTML tag using a query parameter

5. Change your code where you are writing out the HTML H1 element to use the value of a query parameter named `style`. If there is no value for the `style` parameter, use `H1` as the default value.

Test the query parameter

6. Navigate to http://localhost:3000 to see "Hello World" still as an H1 element. Now add your query parameter to change it to a P, for example as in http://localhost:3000?style=p.

Making a client request from node.js

Node.js also has nice support for initiating client requests to an HTTP server.

Run the client

7. Open the file `client.js` in your IDE. This is essentially the same code as we saw in the lecture on client requests. With your web server code still running, run this file. You should see 'Hello World' print out in the console. There are no HTML tags since you added the branching based on the "Accept" header.

Change the request method for the client

8. To make a POST request to the server, change the `method` option from `GET` to `POST`.

Change the content type

9. Go to the `request` object and use `request.setHeader()` to set the value of the "Content-Type" header to "application/json."

Write a JSON object to the request

10. After the header is set, use request.write() to write a JSON object to the request. The object should be an array of strings using the first names of the students around you.

Handling POST requests on the server

Add branching based on the request method in the request listener

11. Go back to the webserver.js file. In the `createServer()` function, replace the `requestListener` function with an inline function that will branch based on the value of the `request.method` property. If the value is 'GET,' call a function named `handleGet()`; if the value is 'POST,' call a function named `handlePost()`. Both functions should have the request and response as parameters.

Create a function for handling GET requests

12. You've already implemented everything for handling GET requests in the `requestListener()` function. Just rename it to `handleGet()`.

Create a function for handling POST requests

13. Create a function name `handlePost()` that takes the request and response as parameters. Add the code to read the data that is in the request using the `IncomingMessage` interface and build a string object with it. Attach a listener for the `'data'` and `'end'` events.

Write the response using the array that was posted to the server

14. When the 'end' event is fired, signaling that you have read all of the data from request, convert the JSON string into an array. Iterate over the contents of the array and build up a string of "Hello [value]" followed by a newline for every element of the array. Then write the string out to the response with a content type of "text/plain."

Test the POST implementation

15. Restart your web server code and then go to the client request and run it with your new POST implementation. You should see "Hello [name]" printed on a new line for every name in the array.

Express

In this lab, you will create a simple "todo" application using the Express web application framework.
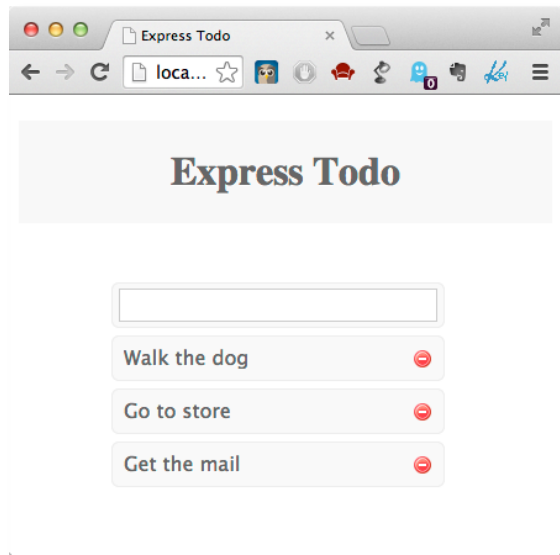
Objectives

In this lab, you will learn to

- define CRUD routes in Express, and
- implement behavior for each route.

Set things up

You have been provided a working Express application to start the lab but first we need to install the required modules.

Install dependencies

1. Run `npm install` in the lab directory.

   This will install the dependencies for our application from the `package.json` file. In this case, our only dependencies are express, an MVC web framework, and jade, a popular template engine that will render our views. The views and basic CSS styling have been provided for you in the `views` and `public` directories.

While not required, you probably want to install `nodemon` (`npm -g nodemon`). `nodemon` automatically restarts your application when a file system change is detected and will save time so you don't have to manually restart Node to view your edits.

Initialize the application

2. Open `app.js` in a text editor and take note of the `fs.readFile` invocation. Normally, a web application would use a database to store data, but working with a database is out of scope for this lab. Instead, we will be using a JSON file, `todo_db.json`, to load sample data into memory. After `todo_db.json` is parsed, we simply store its data in the `app.locals.todos` variable.

(i) *Express provides the variable `app.locals` to store app-level data that is available throughout the application.*

Specify the root route

3. Open `routes/index.js` in your editor and define a HTTP GET route for the root path (`/`) by using `app.get`.

   The first argument to `app.get` is the URL path for the matching route, and the second argument is the callback function that will handle the request. In the spirit of a layered architecture, our controller layer concerning "todos" will be placed in module `todo-controller.js` located in the same directory.

   The service layer will be placed in the `services` directory. You can see that as your application grows, this allows your `routes/index.js` to remain easy to read and keeps the controllers separate. Use `todoController.findAll`, which we'll write in the next step, as the second argument for your route.

Create the controller function for the root route

4. Open `routes/todo-controller.js` and provide an implementation for the function `findAll(req, res, next)`.

   In your implementation, delegate to the `service.findAll` function, and pass to it a callback of the form `function onFound(err, todos)`.

   In the callback, use the `res.render` function to display the `index.jade` view that has already been created for you in the `views` directory.

   As you can see, `views/index.js` expects a variable called `todos` to be passed containing all of the "todo" items, which is used to render edit & delete links for each todo in the browser. The value of todos will come from the service layer call.

(i) *Express is a model-view-controller framework. Controllers are functions (routes) that handle requests, delegate to services, produce models, and select views.*

Views render models. A model is simply a map of data whose entries are used by the view to get to the data. You can think of the model as the "data contract" between the controller and the view: the view expects certain data to be present in the view, and the controller must ensure that it provides that data.

Implement findAll in the service layer

5. Open `services/todo-service.js` and provide an implementation for the function `findAll(callback)`. In a normal application, this would be a database lookup, but in our case, we only need to return `todos`, which has already been passed into the service layer constructor. The implementation should simply be `callback(null, todos)`.

Start the application

6. Run `node app.js` (or `nodemon app.js`) and, if everything is working correctly, you should be able to navigate to and see three sample "todo" items already in your application!

> ⚠️ *You may notice that while our data renders correctly, the rest of the functionality to edit, create, and delete todos is broken because we need to define more routes.*

Create the '/create' route & controller function

7. Open `routes/index.js` and define a HTTP POST route for path `/create` and corresponding function `addTodo(req, res, next)` in `routes/todo-controller.js`.

   According to REST conventions, creation routes are typically defined with a HTTP POST, and our application is no different. The `addTodo` function in `routes/todo-controller.js` should call the service layer's `addTodo` function and pass `req.body.content`, along with a callback of the form `function onAdded(err, todo)`. In the callback, we need to terminate the response, so add `res.redirect('/')` to render the main index view after we've processed the business logic.

Create the addTodo service function

8. Open `services/todo-service.js` and provide an implementation for the `addTodo` function.

   If you examine the `todo_db.json` structure, you will see that we are storing only an `id` and `content`:

   ```
   [{
     "id":"f47b6329-e3db-4af2-e2a1-6fd818ab08ac",
     "content":"Walk the dog"},
     …
   ```

```
}]
```

9. In `services/todo-service.js`, a function to generate unique IDs, called `uuid()`, has been provided for you. Create a new `todo` inline object with property `id` equal to `uuid()` and property `content` equal to the `content` parameter.

   Use the JavaScript `push()` method to add a new entry to `todos`, the array that represents our backing store, simulating database persistence.

   Finally, call the `callback` function with parameters `null`, indicating no error, and the newly created `todo` object.

At this point, you should be able to verify that new items are correctly added in your application. Run your application again via `node` (or `nodemon`) and confirm everything is working ok.

Create the '/edit' route & controller function

In order to render our edit form, we need to define an edit route & corresponding controller function.

10. Open `routes/index.js` and define a HTTP GET route for `/edit` and corresponding controller function `editTodo` in `routes/todo-controller.js`.

    The edit route should be in the form `/edit/:id`, which allows the id parameter to be accessed via `request.params.id`.

    The `edit` view expects a model entry called `todos`, so the `editTodo` function needs to call `service.findAll` and render the `edit` view in the callback, passing the model map.

    The view also expects a `current` model entry, which corresponds to the `id` of the "todo" item to be edited.

If everything works correctly, you should now be able to edit the selected "todo" item. Now, in order to save our changes, we need to define an update route.

Create the '/update' route & controller function

11. Open `routes/index.js` and define a HTTP POST route for `/update` and corresponding controller function in `routes/todo-controller.js`. Like the `/edit` route, `/update` should use a named parameter for the "todo" `id`.

    The `updateTodo` function should call the service layer's `updateTodo` function and pass the updated todo's `id` and `content` for the "todo" item being changed, and a callback of the form `function onUpdated(err)`.

    In the callback, use `res.redirect('/')` to render the root view.

Create the updateTodo service function

12. Open `services/todo-service.js` and provide an implementation for `updateTodo`.

    Loop through all of the `todos`; if the `id` parameter matches the current "todo" item's `id`, update the current todo's `content` property with the `content` parameter, then call `return callback()` to break the loop & return immediately, indicating success.

    If the `id` is not found, then our loop will exit; after the loop, then, invoke the callback with an error.

Create the '/delete' route & controller function

13. Open `routes/index.js` and define a HTTP GET route for `/delete` and corresponding controller function `deleteTodo` in `routes/todo-controller.js`.

    Like the edit and update actions, we need to use a named parameter for the `id` corresponding to the specific "todo" item we wish to remove.

    Call the service layer `deleteTodo` function and pass in the id of the item to be deleted, and a callback of the form `function onDeleted(err)`. Then use `res.redirect('/')` to render the root view in the callback.

> ⚠️ *The HTML specification, as implemented in browsers, does not support the `DELETE` verb in `<form>` elements, so make sure you use a `GET` instead. Express supports all HTTP verbs, including `GET`, `POST`, `PUT` and `DELETE` for RESTful clients.*

Create the deleteTodo service function

14. Open `services/todo-service.js` and provide implementation for `deleteTodo`.

    First, initialize a temporary variable to keep track of an index into the `todos` array. Loop through the `todos`. if the given id matches the current element's id, use `todos.splice(index, 1)` to remove it, then invoke `return callback()` to break the loop and return immediately.

    If the `id` is not found, we will exit the loop; at that point, invoke the callback with an error.

Once everything is working properly, this lab is now complete!

Express Middleware

In this lab, you will use several Express middleware components in the creation of a simple login form.

Objectives

In this lab, you will learn to

- use several built in Express middleware components together, and
- implement custom middleware.

Set things up

You have been provided a minimal Express application to start the lab but first we need to install the required modules.

Install dependencies

15. Run `npm install` in the lab directory.

    This will install the dependencies for our application from the `package.json` file. Our dependencies are `connect`, `cookie-parser`, `cookies`, and `express-session`.

    While not required, you may want to install `nodemon` (`npm -g nodemon`). `nodemon` automatically restarts your application when a file system change is detected and will save time so you don't have to manually restart Node to view your edits.

Add a logger

Open `app.js` in a text editor and you can see that currently our application doesn't do anything aside from listen to port 3000. Let's begin with one of the most common middleware components you will encounter – logging.

Add logger middleware

16. Add the logger middleware to the application with an `app.use()` statement.

    Use the provided `LOGGER` variable to format the log messages by passing it as the argument to `morgan()`. It contains the default logging format plus a couple of additional arguments that we will use later to examine cookies.

Test the logger middleware is working

17. Navigate to http://localhost:3000 and verify that a logging statement appears in the console. The browser will contain an error because we have not yet defined any working routes so let's do that next!

Create a login form

Creating a login form means that our application will need to handle POST form data so next we need to use the `body-parser` middleware.

Add bodyParser middleware

18. Add the `body-parser` middleware after the `logger`. This will ensure that our application and correctly interpret `application/x-www-form-urlencoded` data used in the login form.

Create login middleware

19. Add a custom middleware function called `login` after the `bodyParser` middleware declaration.

    The first thing this middleware should do is check the `req.method` and if it is not `GET`, return `next()`. We will be creating middleware next to handle `POST` data and this middleware should only handle `GET` requests.

20. Create a login form by using `res.setHeader` to set the `Content-Type` to `text/html` and then use `res.write` to generate a HTML form that takes a user and password as inputs and then a submit button. Make sure the input types are `text` for the user and `password` for the password. End the response with `res.end()`.

Create middleware to handle POST data

21. Add a custom middleware function called `checkAuth` and make sure it is declared after the login form. The first thing this middleware should do is check the `req.method` and if it is not `POST`, return `next()`.

    We will create a middleware soon to handle `DELETE` data and this middleware should only handle `POST` requests. The bodyParser middleware allows POST values to be available on req.body. The user and password values should be available as `req.body.user` and `req.body.password`. Create a simple check by validating that `req.body.user === 'admin'` and `req.body.password === 'password'`. If true, use `res.end` to indicate they are logged in and otherwise display an invalid credentials message.

Test the login form

22. Navigate to http://localhost:3000 and make sure that your login form appears. You should be able to see a successful login message with the correct credentials or an error message. This login form isn't very interesting without a session so let's add that next.

Add a session store

Connect supports backing session stores with MongoDB, Redis, and others. In this lab we will just be using the default memory store.

Add session middleware

23. After the `bodyParser` middleware, add the `cookieParser()` middleware and then `session()` middleware. The session middleware requires a secret to be passed in the format `app.use(session({ secret: 'super secret string' }))`.

Add session regeneration in checkAuth function

24. In the `checkAuth` function, add `req.session.regenerate(function createSession(err) {…})` if there is a successful login. Move the successful login message inside this callback.

> (i) *A normal web application would redirect to some other location after a successful login but redirect functionality is not built into Connect. Express, a higher-level framework built on top of Connect, provides this functionality for more traditional web development.*

Add logout middleware

At this point, we can login successfully with the application and create a session but we don't have a way yet to logout and destroy the session.

Add methodOverride middleware

25. Add the `methodOverride` middleware to the application after the `bodyParser`. This middleware will allow us to send a form with a `DELETE` method and Connect will populate the `req.method` object.

Add a logout form to the login middleware

26. Modify the login middleware by checking to see if `req.session.user` exists. If so, present a form to logout. Inside the `form` element use the `action` attribute to pass `_method=DELETE` as a query parameter. If `req.session.user` does not exist, present the login form as before.

Add logout middleware

27. Create a custom middleware and place it last in the chain of middleware functions you have defined so far. Check that `req.method` is equal to `DELETE` and then use `req.session.destroy(function logout(err) {…})` to destroy the session. Inside this callback, use `res.end` with a message indicating the user is logged out.

Test the application

28. Navigate to http://localhost:3000 and verify that the ability to log in and out of the session works properly.

Once everything is working properly, this lab is now complete!

RESTful APIs with Express

In this lab, you will build upon the "todo" application created in the previous lab and create a REST API alongside the web interface.

Objectives

In this lab, you will learn to

- define RESTful routes in Express, and
- implement behavior for each route.

Set things up

You have been provided a working Express application to start the lab but first we need to install the required modules.

Install dependencies

X. Run `npm install` in the lab directory. This will install the dependencies for our application from the `package.json` file. In this case, our only dependency is `express`, an MVC web framework for Node.js.

While not required, you probably want to install `nodemon` (`npm -g nodemon`). `nodemon` automatically restarts your application when a file system change is detected and will save time so you don't have to manually restart Node to view your edits.

Specify the findAllJson route

X. Open `routes/index.js` in your editor and define a HTTP GET route for the REST path to get all todos, `/api/todos`, by using `app.get`. The first argument to `app.get` is the URL path for the matching route, and the second argument is the callback function that will handle the request. In the spirit of a layered architecture, our controller layer concerning todos will be placed in module `todo-controller.js` located in the same directory. The service layer will be placed in the `services` directory. You can see that as your application grows, this allows your `routes/index.js` to remain easy to read and keeps the controllers separate. Use `todoController.findAllJson`, which we'll write in the next step, as the second argument for your route.

> ⓘ *Our REST API will be prefixed with `/api` so there is no confusion about which routes belong to the web layer and which belong to our API.*

Create the REST endpoint to get all todos

X. Open `routes/todo-controller.js` and provide an implementation for the function `findAllJson(req, res, next)`. In your implementation, delegate to the `service.findAll` function, and pass to it a callback of the form `function onFoundJson(err, todos)`. The `service.findAll` function has already been implemented from the previous lab. In the callback, use the `res.json(todos)` function to send a JSON response containing all todos. If there is an error, return it with `res.json({message : err.message})`.

> ℹ️ *`response.json()` is identical to `response.send()` when an object or array is passed. It may also be used for explicit JSON conversion of non-objects (null, undefined, etc.).*

> ⚠️ *In the event of an error, be sure to always send a JSON response containing your error message and any other important information (stack trace, error code, etc) rather than sending a response code indicating failure like 500. Sending error information as a JSON response is a best practice to give consumers of your API as much help as possible to diagnose a problem.*

Start the application and test the REST endpoint to get all todos

X. Run `node app.js` (or `nodemon app.js`). Open a separate terminal window and issue the command `curl -i http://localhost:3000/api/todos`. If everything is working correctly, you should see a JSON response with three sample todo items.

> ℹ️ *If you are able to run Google Chrome on your lab machine and would prefer to use a graphical REST client, please refer to Appendix A.*

Create the REST endpoint to create a todo

X. Open `routes/index.js` and define a HTTP POST route for path `/api/todos` and function `addTodoJson(req, res, next)` in `routes/todo-controller.js`.

> ℹ️ *REST conventions dictate that the HTTP method GET is used for reading resources, POST for creating, PUT for updating, and DELETE for deleting.*

The `addTodoJson` function in `routes/todo-controller.js` should call the service layer's `addTodo` function and pass `req.body.content`, along with a callback of the form `function onAddedJson(err, todo)`. In the callback, use `res.json(todo)` to send a JSON response

containing the newly added todo item. If there is an error, return it with `res.json({message : err.message})`.

Test the creation endpoint

X. Test the `addTodoJson` route by issuing a command in the form `curl -i -X POST -H "Content-Type: application/json" -d '{ "content" : "Get the milk" }' http://localhost:3000/api/todos`. The new todo item should be returned in a JSON response. You can also invoke the REST endpoint GET '/api/todos' to see the entire list with your new item added.

Create the REST endpoint to get a single todo by id

X. Open `routes/index.js` and define a HTTP GET route for `/api/todos/:id` and corresponding controller function `findOneJson` in `routes/todo-controller.js`. The token `:id` in the route is a named route parameter that allows the id parameter to be accessed via `request.params.id`. The `findOneJson` function should call the service layer's `findOne` and pass the todo's `id` and a callback of the form `function onFoundJson(err, todo)`. The service layer function `findOne` has already been provided for you. In the callback, use `res.json(todo)` to send the JSON response. If there is an error, return it with `res.json({message : err.message})`.

Test the REST endpoint to get a single todo

X. Test the `findOneJson` route by issuing a command in the form `curl -i http://localhost:3000/api/todos/508d4a36-3c7c-7bb3-c5ee-1c4c3fb440f6`. The todo item specified by the id should be returned.

Create the REST endpoint to update a todo

X. Open `routes/index.js` and define a HTTP PUT route for `/api/todos/:id` and controller function `updateTodoJson` in `routes/todo-controller.js`. Like the `findOneJson` route, `updateTodoJson` should use a named route parameter for the todo `id`. The `updateTodoJson` function should call the service layer's `updateTodo` function and pass the updated todo's `id` and `content` for the item being changed, and a callback of the form `function onUpdatedJson(err, todo)`. In the callback, use `res.json()` to return the updated todo's `id`. If there is an error, return it with `res.json({message : err.message})`.

Test the REST endpoint to update a todo

X. Test the `updateTodoJson` route by updating the "Get the mail" todo to be "Pick up pizza" instead. Issue the command `curl -i -X PUT -H "Content-Type: application/json" -d '{ "content" : "Pick up pizza" }' http://localhost:3000/api/todos/508d4a36-3c7c-7bb3-c5ee-1c4c3fb440f6`. A JSON response of the updated todo's `id` should be returned.

Create the REST endpoint to delete a todo

X. Open `routes/index.js` and define a HTTP DELETE route for `/api/todos/:id` and controller function `deleteTodoJson` in `routes/todo-controller.js`. Like the `findOneJson` and `updateTodoJson` routes, we need to use a named route parameter for the `id` corresponding to the

specific todo item we wish to remove. Call the service layer `deleteTodo` function, pass in the id of the item to be deleted, and a callback of the form `function onDeletedJson(err)`. If there is no error, use `res.json()` to return the id of the deleted item. If there is an error, return it with `res.json({message : err.message})`.

Test the REST endpoint to delete a todo

X. Test the `deleteTodoJson` route by issuing a command in the form `curl -i -X DELETE http://localhost:3000/api/todos/508d4a36-3c7c-7bb3-c5ee-1c4c3fb440f6`. If the command is successful, the id of the deleted item should be returned.

Once everything is working properly, this lab is now complete!

Appendix A

Google Chrome

If your lab machine is equipped with Google Chrome, you may find it easier to work with a GUI based REST client rather than the `curl` command line utility. There are a number to choose from in the Chrome Web Store. One client that has worked well for the lab authors is Postman: http://www.getpostman.com.

Async.js

In this lab, you will familiarize yourself with the Node module Async.js, which assists in various ways to perform parallel & serial control flow as well as to iterate collections in both parallel & serial modes.

Objectives

You will learn to use `async.auto` to more cleanly control program flow among interdependent functions.

Install Async.js

The first thing we have to do is to use Node Package Manager, `npm`, to install the `'async'` module. Do this by opening a command prompt in the lesson directory and issuing the command `npm install async`. This will download & install the latest version of the Async.js module into a directory called `node_modules`.

Lab Steps

Now that Async.js is installed, we can begin the lab, whose goal is to use a simple (and admittedly buggy) minification algorithm to reduce the size of a JavaScript file.

Add tasks

Since `async.auto` is a function that takes an object representing the tasks to be performed and a callback to be invoked when all tasks have been completed, we need to fill in the stubbed out tasks variable in `auto.js` with our initial requirements.

Notice that we've defined several functions: `exists`, `read`, `create`, `write` & `close`. These all deal with the JavaScript file we're reading from and the minified JavaScript file we're writing to. The functions' interdependencies are the following:

- `read` must be called only after `exists` is successfully called,
- `write` must be called only after `read` & `create` are successfully called, and
- `close` must be called only after `write` is successfully called.

Let's start slowly, and simply add our first function to the tasks object that calls `exists`.

1. Add a property named `exists` to variable `tasks` whose value is a reference to the `exists` function.

Invoke async.auto

2. Now that we've got a minimal tasks object created, add the invocation of `async.auto` that takes the `tasks` object and a callback of the form `function(err, results)` that checks for the existence of an error and, if one exists, logs it to the console, otherwise simply logs a message that the file was minified ok.

3. Execute your tiny workflow consisting simply of the call to `exists` by executing `node auto.js` at a command prompt.

Once you see your `'minified ok'` message, move on to the next step.

Add the read task

The next thing to do is to add to the workflow the call to the read function, but only after the call to `exists` has completed successfully.

4. Update your `tasks` object to contain a property called `read` whose value is an array of a string containing the name of the key you used for the task that confirms file existence (`exists`) and a reference to the `read` function.

5. Execute your workflow again via `node auto.js` at the command prompt. This time, you should observe your `exists` function being invoked before the `read` function.

Once you see the order of invocations that you expect and your `'minified ok'` message again, move on to the next step.

Add the create task

Before we can write anything to the destination file, let's add a call to the `create` function, which ensures that the file is created and opened with a file descriptor. Note that `create` has no prerequisites because we're always creating & overwriting the destination file.

6. Update the `tasks` object with a property `create` whose value is a reference to the create function.

7. Execute your workflow again via `node auto.js`.

Once you see your `create` function being called and your `'minified ok'` message, move on to the next step.

Add the write and close tasks

We're now going to finish our minification workflow by adding two more tasks: one to write the contents of the source file and one to close the destination file after it's been written.

Notice that the `buffer` we're going to write to the file is set to `null` initially. Your job is to create a new `Buffer` whose contents are equal the array-ified string contents read from the `read` function. Where can you find that? Thanks to `async.auto`, we can find it in the `results` object that's given to the `write` function!

You see, `async.auto` tracks all values returned by tasks and stores them as properties on a `results` object that is given to any function that wants them (including the final completion callback given to `async.auto` itself).

8. Set the `buffer` to a new `Buffer` object whose value is simply `results.read.join(' ')`, which is the result of the elements of the array returned by the `read` function (`results.read`) concatenated with a space (`join(' ')`).

9. Set the file descriptor `fd` to `results.create`, which is the file descriptor returned by the function `create`.

Lastly, we want to be good resource citizens and close the destination file.

10. Implement the `close` function by invoking `fs.close` using the same file descriptor used above.

11. Now, update the tasks object to include a `write` property that depends on `read` and `create` & invokes `write`, and a `close` property that depends on `write` & invokes `close`.

12. Execute `node auto.js`, which now has our complete workflow, and observe each function invocation happening only after each function's prerequisites have been satisfied.

Once you see the correct invocations and your 'minified ok' message, you have completed this lab!

Yeoman: Scaffolding Node Applications

In this lab, you will familiarize yourself with Yeoman, a suite of tools that enable quick scaffolding, in-browser dependency management, and mature builds.

Objectives

You will learn to use the following tools:

- `yo`, a Node.js scaffolding plug-in framework,
- `bower`, a dependency management tool for the front-end, and
- `grunt`, a generic JavaScript task runner that is used to build Node applications.

Install Tools

The first thing we have to do is to use Node Package Manager, `npm`, to install `yo`. Do this by opening a command prompt in the lesson directory and issuing the command `npm install -g yo`. This will download & globally install the latest version of `yo`. This will automatically install the tools `bower` & `grunt`. Last, install the `yo` generator for Angular.js with `npm install -g generator-angular`.

Lab Steps

Now that the Yeoman tools are installed, we can begin the lab, whose goal is to scaffold an Angular web application, then add our own controller, view & route.

Create the web application directory

1. Change to this lesson's lab directory and create a directory in it called `todo` with `mkdir todo`.

Scaffold the initial web application

2. Change to the `todo` directory you just created and scaffold the initial Angular application with the command `yo angular`.

⚠️ *Choose **not** to use Compass, and then take the defaults for the remaining prompts.*

At this point, you should have a directory structure that looks similar to this:

```
Name
▼ 📁 todo
      📄 .bowerrc
      📄 .editorconfig
      📄 .gitattributes
      📄 .gitignore
      📄 .jshintrc
      📄 .travis.yml
   ▼ 📁 app
         📄 .buildignore
         📄 .htaccess
         📄 404.html
      ▶ 📁 bower_components
         📄 favicon.ico
      ▶ 📁 fonts
      ▼ 📁 images
            📄 yeoman.png
         📄 index.html
         📄 robots.txt
      ▼ 📁 scripts
            📄 app.js
         ▼ 📁 controllers
               📄 main.js
      ▼ 📁 styles
            📄 main.css
      ▼ 📁 views
            📄 main.html
      📄 bower.json
      📄 Gruntfile.js
      📄 karma-e2e.conf.js
      📄 karma.conf.js
   ▶ 📁 node_modules
      📄 package.json
   ▼ 📁 test
         📄 .jshintrc
         📄 runner.html
      ▼ 📁 spec
         ▼ 📁 controllers
               📄 main.js
```

Run the web app

3. Since `yo` integrates with `bower` & `grunt`, we can run the web app right out of the box. Do so with the command `grunt serve`.

> **(i)** *As part of scaffolding the initial web app, `yo` defines a custom task called `serve` that builds & runs your application in a new browser window with LiveReload so that edits are immediately reflected in the browser.*

You should see a new browser window pop up with the address http://127.0.0.1:9000/#/ with a view that looks like this:

todo   **Home**   About   Contact

Hello

YEOMAN

Always a pleasure scaffolding your apps.

Splendid!✔

HTML5 Boilerplate

HTML5 Boilerplate is a professional front-end template for building fast, robust, and adaptable web apps or sites.

Angular

AngularJS is a toolset for building the framework most suited to your application development.
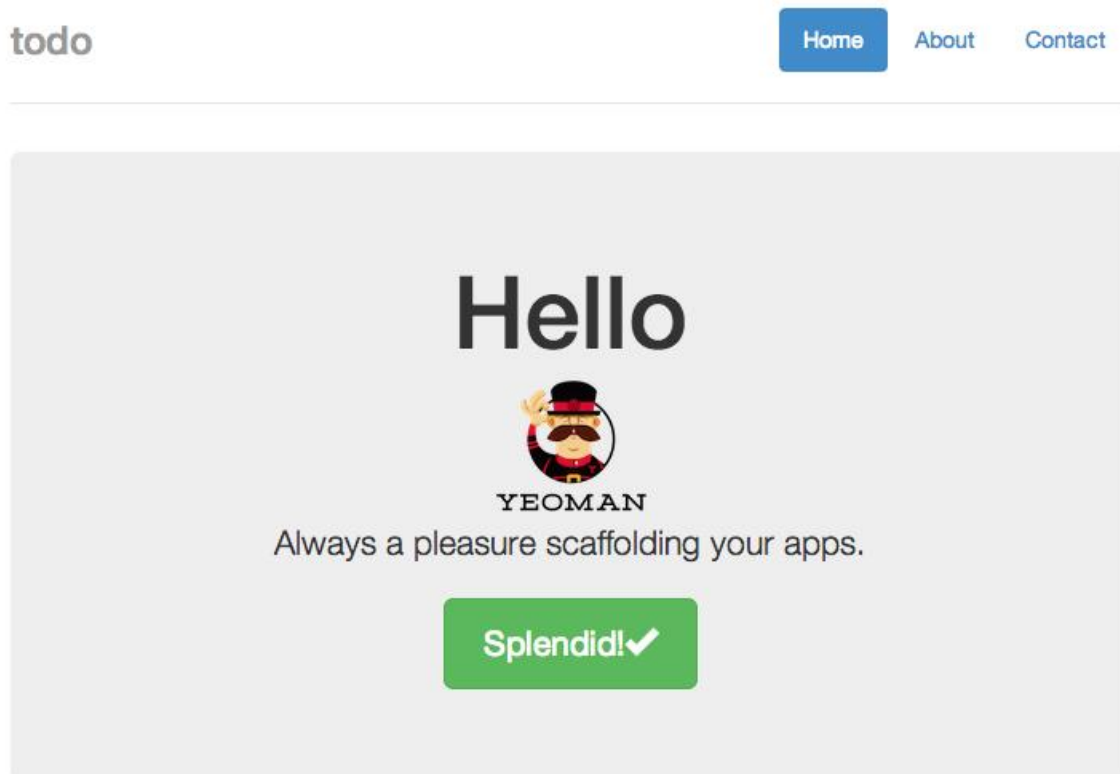
Karma

Spectacular Test Runner for JavaScript.

♥ from the Yeoman team

Make a live edit
Now, let's see how editing is reflected immediately via LiveReload.

4. Open the file `todo/app/views/main.html` and change the text in it from `<h1>'Allo,` `'Allo!</h1>` to `<h1>Hello</h1>`. You should see the new content in the browser without having to refresh the page:

Use model from controller in view
The page above is primarily statically rendered. Let's change this by using the model provided by the controller in the view.

5. Open the file `todo/app/scripts/controllers/main.js` and notice that the controller is populating the `$scope` with a property called `awesomeThings` with an array of three strings.

*Angular routes requests to **controllers**, which produce **models**, which are then available to **views**. A controller places data to be rendered by a view by adding properties to the provided `$scope` service, given in a controller's constructor function.*

For example, in `app/app.js`, you see the route for the root URL:

```
    $routeProvider
      .when('/', {
        templateUrl: 'views/main.html',
        controller: 'MainCtrl'
      })
```

6. Update the view `app/views/main.html` to use the model that's produced by controller `'MainCtrl'` by changing the content to nothing more than

```
<div class="container">
  <h2>Awesome Things</h2>
  <ul>
    <li ng-repeat="thing in awesomeThings">{{thing}}</li>
  </ul>
</div>
```

The view should be updated immediately to reflect the new view, which now dynamically renders the model placed in `$scope` by the controller. When you see this view, move on to the next step.

Create a new single page application

7. Now, use the scaffolding offered by `yo` to create a new page ("single page application"), route, controller & view by opening a new terminal, changing to your `todo` directory, then issuing the command `yo angular:route square`. The artifacts created are listed below.

First, the route: `app/app.js` has been updated with the new route

```
.when('/square', {
  templateUrl: 'views/square.html',
```

```
    controller: 'SquareCtrl'

})
```

Next, the initial controller implementation in `app/scripts/controllers/square.js` simply returns some sample data (the same as controller `'MainCtrl'`):

```
angular.module('todoApp')

  .controller('SquareCtrl', function ($scope) {

    $scope.awesomeThings = [

      'HTML5 Boilerplate',

      'AngularJS',

      'Karma'

    ];

  });
```

Last, the default view in `app/views/square.html` is simply

```
<p>This is the square view.</p>
```

If you now visit `http://127.0.0.1:9000/#/square`, you'll see the simple view.

Update the controller and view to render squares of a few numbers

Now, we're going to update the controller and view so that it renders the squares of a few numbers, but first, we need to abide by good software development principles and ensure that we're testing our controller. Fortunately, grunt provides us with automatic unit and end-to-end testing capabilities.

Install unit test modules

Before our tests will execute, make sure to install the test modules that we'll need and include them in our development environment's dependencies.

8. Kill the `grunt` process (via `Ctrl-C`), then execute `npm install karma-jasmine --save-dev` and `npm install karma-chrome-launcher --save-dev`. This will add these modules to the file `todo/package.json`.

9. Restart the server via `grunt serve`, then point the browser at `http://127.0.0.1:9000/#/square`. You should see the simple square view that you did before.

Write a unit test for the controller

10. Open file `todo/spec/controllers/square.js`, the file that holds unit tests for the controller `'SquareCtrl'`, and update the test's `it` function to be the following:

```
it('should attach a list of squares of the form { val: n, squared: n*n }
where n is from 1 to 6 to the scope', function () {

    expect(scope.squares.length).toBe(6);


    [1, 2, 3, 4, 5, 6].forEach(function (elem) {

      var obj = scope.squares[elem - 1];

      expect(obj).toEqual({

        val : elem,

        squared : elem * elem

      });

    });

  });
```

This places the expectation on our controller to place the expected model into scope.

Once you save this file, you should see an error from grunt, stating something similar to the following:

```
Chrome 34.0.1847 (Mac OS X 10.9.2) Controller: SquareCtrl should attach a
list of squares of the form { val: n, squared: n*n } where n is from 1 to 6
to the scope FAILED
```

```
    Expected 0 to be 6.

    …
```

You're seeing this error because our generated controller doesn't meet the specification that we just declared above.

⚠️ *You may need to touch the test file,* `todo/test/spec/controllers/square.js`, *in order to get grunt to rerun the tests.*

11. Update the controller in `app/scripts/controllers/square.js` to place an array of objects with two properties each: `val`, containing a number, and `squared`, containing the square of the value. Since you're probably as lazy as the author of this lab is and writing code to do this is not the focus of the lab, here's the code.

```
angular.module('todoApp')

  .controller('SquareCtrl', function ($scope) {

    $scope.squares = [];

    var vals = [1, 2, 3, 4, 5, 6];

    vals.forEach(function onEach(it) {

      $scope.squares.push({

        val: it,

        squared: it * it

      });

    });

  });
```

Now, you should see a line similar to the following in your terminal, indicating successful testing (again, you may need to touch the test file again):

```
Chrome 34.0.1847 (Mac OS X 10.9.2): Executed 2 of 2 SUCCESS (0.027 secs /
0.025 secs)
```

12. Update the view `app/views/square.html` to render the values of the object at the key `'squares'` in the model (which happens to be an array). Again, here's the code:

```
<p ng-repeat="sq in squares">{{sq.val}}<sup>2</sup> = {{sq.squared}}</p>
```

Now, view `http://127.0.0.1:9000/#/square` in a browser. You should see something similar to this:

$$1^2 = 1$$

$$2^2 = 4$$

$$3^2 = 9$$

$$4^2 = 16$$

$$5^2 = 25$$

$$6^2 = 36$$

> ⚠️ *It is important to note that the model forms the contract between the controller and the view. The controller is responsible for populating the model, and the view must know the shape of the model so that it can render it properly. Any changes to the view that require new data to be added to the model must be accommodated by the controller so that the controller provides that data.*

When you see the expected output, you have completed the lab!

Dust

In this lab, you will transform a static HTML page into a Dust template.

Objectives

In this lab, you will

- extract a model from a static HTML page,
- use Dust logic helpers and section tags to navigate your model,
- transform a static HTML page into a Dust template.

Run Dust with Node

For this lab, we will be running Dust from start to finish in Node. In your lab directory you will see a file named `rundust.js`. This application will compile your template, load it into Dust, and render an HTML page into a file.

1. Execute the following command in a shell:

```
node rundust quiz.dust
```

This will create a one-line html file, `quiz.html`. Rerun this command as needed as you are developing your template to see your results.

Open your original, static HTML page

2. Open the file `original-quiz.html` in an editor. This is your starting point. Open it up in a browser also to see what it looks like rendered. This is a very simple page for gathering up students' answers to a set of assessment questions. Consider the pieces that should be taken out of this HTML so that the finished template can be used with any assessment.

Open the template and model files

3. Open the file `quiz.dust` in an editor. This is where you will build your template

4. Open the file `model.js` in an editor. This is where you will build your model.

Design your data model

5. Perhaps the easiest way to start is to just copy the contents of the static HTML page into your template. Then look at the information in the HTML and begin designing your data model. You should wind up with three major types: the assessment as a whole, a question, and an answer. Shell out the model in your model file.

Populate the model

5. Extract the information from the HTML that belongs in the model and not the page. Replace any values you want with placeholder tags using the property names from your model.

Create the template logic

5. There are a number of places where you will want to use the easy iteration of arrays built into Dust sections. There's also a lot of similarity between a set of radio button inputs and a checkbox input. Consider using a logic helper.

There is no "right" solution to this lab. There is a tremendous amount of flexibility in Dust. Have fun coming up with surprises!

When you're finished page matches the original, this lab is complete!

KrakenJS

In this lab, you will create a simple "todo" application using the Kraken web application framework.

Objectives

In this lab, you will learn to

- create a model, view, and controller for "todo" items in Kraken,
- implement behavior for each route, and
- create an internationalized title.

Install Tools

Before starting the lab, you will need a MongoDB server running on your lab machine for the database backend in our application. You will also need to install the `generator-kraken` module that will be used to generate a skeleton Kraken application.

Install MongoDB

29. Go to the MongoDB homepage at http://www.mongodb.org and install the latest version for your operating system.

30. Once installed, start the server with the `mongod` command line utility.

Install generator-kraken module

31. If you did not install Yeoman from the previous lab, you should do this now by opening a command prompt in the lesson directory and issuing the command `npm install -g yo`. This will download & globally install the latest version of `yo`.

32. Open a command prompt in the lesson directory and issue the command `npm install -g yo generator-kraken`. It may be necessary to prefix the `npm` command with `sudo` if you are using a Mac or Linux platform.

Lab Steps

Now that the tools are installed, we can begin the lab.

Scaffold a new web application with Kraken

33. Open a terminal in the lab directory, then issue the command `yo kraken`.

34. Enter `todo` for the app name, `Todo Application` for the description, and whatever you want for the author.

35. Take the defaults for the remaining prompts.

This will scaffold a web application that uses Express and Kraken in the directory `todo`.

Start & view the web application

36. Change into the `todo` directory and issue the command `npm start`; this will start the server process.

37. Open `http://localhost:8000` in a web browser. You should see something similar to the following:

Hello, index!

Once you see this, move on to the next step.

Add MongoDB connectivity

38. Open `package.json` in your editor and add `"mongoose": "3.8.12"` to the `dependencies` section.

39. Copy the `lib` directory from the `lab` directory of this lesson into the `todo` directory, which contains your application.

40. Open `index.js` in your editor and add the following require statement at the top: `var db = require('./lib/db');`.

41. Add the line `db.config(config.get('databaseConfig'));` in the function at `options.onconfig` of `index.js`, right after the comment "any config setup/overrides here."

    Make sure `next(null);` is the last statement in the function.

42. Open `config/config.json` and add the following section to this file immediately before the `middleware` property:

    ```
    "databaseConfig": {
      "host": "localhost",
      "database": "test"
    }
    ```

43. Stop the server and run `npm install` to install the mongoose dependency for MongoDB connectivity.

44. Start the server again with `npm start` and you should now see a message in the console that says "db connection open" indicating the MongoDB connection is working.

Generate a controller and dependencies

45. Stop the server and run `yo kraken:controller todos` which will generate a model, view, controller, and I18N bundle in their respective places.

    Accept the default at the prompt.

Edit model

46.   Open `todo/models/todos.js` and replace it with the contents of `impl/model-todos.js`. You can also just replace the file, but this will give you a chance to review the code.

This defines a very simple "todo" model with mongoose that only contains one `String` field.

Edit controller

47. Open `todo/controllers/todos/index.js` and replace it with the contents of `impl/controller-todos.js`.

This controller defines typical create, read, update, delete routes using that should be familiar to most web developers.

Copy CSS styles and graphics

48. Copy `lab/css/app.less` over the existing `todo/public/css/app.less`. This is a simple stylesheet and delete button similar to what was used in the Express lab.

49. Create the directory `todo/public/png`.

50. Copy `lab/png/delete.png` to `todo/public/png/delete.png`.

Edit views

51. Open the file `todo/public/templates/layouts/master.dust` and add the following line as the last line in the `head` element:

    ```
    <link rel="stylesheet" href="/css/app.css">
    ```

52.   Open `todo/public/templates/todos/index.dust` and replace it with the contents of impl/todos.dust.

This Dust template first contains a form for creating new "todo" items. Then it checks for the existence of a `todos` object to render the list of all "todos". If the `todos` object does not exist, it checks for the `update_todo` object that is used to render the item to be updated. If neither exists, there are no "todo" items in the database yet.

Run the web application

53. Start the web application again with `npm start`, return to the browser and view the URL `http://localhost:8000/todos`. You should see the following screen:

Verify that you are able to

create, edit, and delete "todo" items and then move on to the next step.

Internationalize the title

For the final part of the lab, we will take a brief look at how you can internationalize strings with Kraken.

Create setLanguage controller

54. Create a file called `controllers/setLanguage/index.js` and paste the following implementation:

```
'use strict';

module.exports = function (server) {
  server.get('/setlanguage/:lang', function (req, res) {
    res.cookie('language', req.param('lang'));
    res.redirect('/todos');
  });
};
```

This route will set a cookie called `language` depending on the named parameter. We will add the language choices in the Dust template soon.

Add language middleware

55. Open `config/config.json` .

```
"language": {
    "priority": 95,
    "enabled": true,
    "module": {
         "name": "path:./lib/language"
    }
},
```

Open `todo/lib/language.js` and examine what is happening. This middleware checks for the `req.cookies.language` cookie and if it exists, `res.locals.context.locality` is set with its value. This will cause Kraken to use the appropriate language bundle when rendering the template.

Add a Spanish language bundle

56. Create `locales/ES/es/todos/index.properties` and paste the following key/value:
    greeting=Desata el Kraken!

57. Edit `locales/US/en/todos/index.properties` and change the `greeting` key to Unleash the Kraken!

Modify template to switch languages

58. Open `public/templates/todos/index.dust` and make the following modification:

```
…
{<body}
  <p>Select language:</p>
  <p><a href="/setLanguage/en-us" alt="English">English</a><br/>
  <a href="/setLanguage/es-es" alt="Spanish">Spanish</a></p>
  <h1 id="page-title">{@pre type="content" key="greeting"/}</h1>
  <div id="list">
…
```

This will create two links from which the `setLanguage` controller can be called and the appropriate cookie set. The title has been modified to use the `greeting` key from the message bundle instead of a hard coded greeting.

Run the web application

59. Start the web application again with `npm start`, return to the browser and view the URL `http://localhost:8000/todos`. You should now see two links available and see something similar to the following if you select Spanish:

Select language:

English
Spanish

Desata el Kraken!

Walk the dog

Learn kraken.js

Once everything is working properly, this lab is now complete!

Security in Kraken with Lusca

In this lab, you will familiarize yourself with the security features in Kraken, offered by Lusca.

Objectives

You will learn how to prevent

- cross-site request forgeries (CSRF/XSRF),
- cross-site scripting attacks (XSS),
- a browser from downloading unauthorized content via content security policies (CSP).

Install Tools

The first thing we have to do is to use Node Package Manager, `npm`, to install `generator-kraken`. Do this by opening a command prompt in the lesson directory and issuing the command `npm install -g generator-kraken`.

Lab Steps

Now that the tools are installed, we can begin the lab, whose goal is to scaffold a Kraken web application, then use Lusca to add our security features for CSRF & CSP.

Scaffold a new web application with Kraken

1. Open a terminal in the lab directory, then issue the command `yo kraken`. Enter `lusca-test` for the app name, `Lusca Test Application` for the description, and whatever you want for the author. Take the defaults for the remaining prompts.

This will scaffold a web application that uses Express and Kraken in the directory `lusca-test`.

Start & view the web application

2. Change into the `lusca-test` directory and issue the command `npm start`; this will start the server process.

3. Open `http://localhost:8000` in a web browser. You should see something similar to the following:

Hello, index!

Once you see this, move on to the next step.

Protect the web application from cross-site request forgery

Out of the box, our web application is susceptible to cross-site request forgery (CSRF or XSRF). In order to prevent this, let's configure this protection now.

4. Open `lusca-test/config/middlware.json` and notice that property `appsec` is commented out. Uncomment it, remove the lines for `p3p` & `xframe` (since we're not addressing these in this lab), comment the `csp` property, which we'll be addressing in the next step, and ensure that the `csrf` property is set to `true`.

At this point, our web application is now no longer susceptible to CSRF attacks. Let's write some code now to prove it.

Add a controller to test CSRF prevention

5. Kill the web server (`Ctrl-C`) and issue the command `yo kraken:controller csrf`. This will create a controller at `lusca-test/controllers/csrf.js`, a model at `lusca-test/models/csrf.js`, and a Dust.js view template at `lusca-test/public/templates/csrf.dust`.

Update the view to demonstrate CSRF prevention

6. Open the Dust template at `lusca-test/public/templates/csrf.dust` and change its content to the following:

```
{>"layouts/master" /}

{<body}

  <h1>{@pre type="content" key="greetings"/}</h1>

    <h2>{@pre type="content" key="form"/}</h2>

    <form method="post">

      Name: <input type="text" name="name" value="{name}" size="100"/>

      <br/>

      CSRF Token: <input type="text" name="_csrf" value="{_csrf}" size="100"/>

      <br/>

      <input type="submit" value="Submit"/>

    </form>
```

```
{/body}
```

Take a moment to look at this form. It renders a simple form that includes an input for a name and an input showing the current CSRF token that is used to prevent CSRF attacks.

⚠️ *Normally, this value is used in hidden form inputs, as it's not intended for human consumption.*

Update the i18n CSRF bundle

7. Notice the `<h2>` element in the template; it's expecting a message keys called `greetings` and `form`. Open `lusca-test/locales/US/en/csrf.properties` and replace its content with the following:

```
form=Enter name, then click submit.  Change the CSRF token to force error.

greetings=Hello, {name}.
```

Update the default CSRF model

8. Open the model file `lusca-test/models/csrf.js` and change it to return the literal JavaScript object `{ name: 'Dude' }`, which will be the default model that our controller will have the view render.

Update the CSRF controller

9. Open the controller in `lusca-test/controllers/csrf.js` and change its content to export a function that handles HTTP GETs & POSTs:

```
module.exports = function(app) {


  var model = new CsrfModel();


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

    res.render('csrf', model);

  });



  app.post('/csrf', function(req, res) {

    model.name = req.body.name || '(none)';

    res.render('csrf', model);

  });
```

As you can see, GETs will cause simply cause the form to be rendered will the default model, and POSTs will update the model with the name given in the `name` input from the HTML form, or `(none)` if one wasn't entered. So far, so good. Easy peasy!

Run the web application

10. Start the web application again with `npm start`, return to the browser and view the URL `http://localhost:8000/csrf`. You should see the form looking similar to this:

Hello, Dude.

Enter name, then click submit. Change the CSRF token to force error.

Name: `Dude`
CSRF Token: `SVzK6c3JdKMWmln6Xs4rambQ6sHJ5wGX4KiDM=`
Submit

Notice the CSRF token. This changes on each request to ensure that only authorized requests are coming into this web application. Go ahead and refresh the form and notice that the CSRF token changes each time.

When you see the above form and the new tokens upon each GET request, move on to the next step.

Change the name and view the result

11. Enter a new name in the `name` field and view the result. You should see something similar to the following:

Hello, New Name.

Enter name, then click submit. Change the CSRF token to force error.

Name: New Name
CSRF Token: swOAmhBtJcTn6kiw1AElj5diqrDfsNJUx087A=
Submit

The site successfully handled the POST and rendered a new model. When the site is behaving properly, move on to the next step.

Elicit a CSRF attack prevention

Now, we're going to simulate a CSRF attack. An attacker cannot know the CSRF token when attempting a request from a third party machine; the CSRF can only come from the origin web server, and is cryptographically secure.

12. Before the next form submittal, change the value of the CSRF token (or delete it), simulating an attack. The resultant page should look something like the following:

Internal server error

The URL `/csrf` had the following error `Error: CSRF token mismatch.`

Our CSRF middleware component, configured in `lusca-test/middleware.js`, has detected the unauthorized request and denied it! Issue a new GET request to `http://localhost:8000/csrf` and resubmit the form without changing the CSRF token; you should see a successful result.

When you see the site behaving correctly, move on to the next step, knowing that the browser can now only respond to requests that originated from this server.

Introduce a cross-site scripting (XSS) vulnerability

Next, we're going to see how a site can open a hole for an attacker to issue a cross-site scripting (XSS) attack.

12. Stop the web server and issue the command `yo kraken:controller echo`, which will generate a model, controller view, and i18n bundle in their respective places. As you might guess by the name, we're going to simply echo a string that the user submits via a web form.

Implement the echo functionality

Now, let's implement the echo functionality.

13. First, let's modify the generated echo model, by replacing the content of lusca-test/models/echo.js with the following:

```
module.exports = function EchoModel() {

    return {

        message: 'Enter a message'

    };

};
```

14. Next, update the controller to handle HTTP GET & POST requests by simply populating the model with the value of the `message` variable:

```
module.exports = function (app) {


    var model = new EchoModel();


        function handle(req, res) {

            model.message = req.param('message') || model.message;

            res.render('echo', model);
```

```
        }


    app.get('/echo', handle);

    app.post('/echo', handle);

};
```

15. Now, update the Dust template `lusca-test/public/templates/echo.dust` that is the HTML form page to have the following content:

```
{>"layouts/master" /}


{<body}

    <h1>{@pre type="content" key="greeting"/}</h1>

    <form method="post">

      Message: <input type="text" name="message" value="{message}"
size="100"/>

       <input type="hidden" name="_csrf" value="{_csrf}"/>

       <input type="submit" value="Submit"/>

    </form>
{/body}
```

As you can see, it's a simple form with an input that will be echoed in the `<h1>` element.

16. Update the i18n properties to introduce the XSS vulnerability by changing its content to the following:

```
greeting=You entered: {message|s}
```

Notice the trailing |s in the curly braces; this instructs Dust.js not to do any escaping of the content, instead rendering the content as given. *This, by the way, is the key to the XSS vulnerability.*

Start the vulnerable site

17. Restart the website with `npm start` and test that the form is behaving properly; that is, that it is echoing the contents of the `message` form variable.

Now, it's time to put on our attacker hat. Try entering a message value that includes some HTML, like `Boo <em>Hoo!</em>` and notice what happens when its rendered. You should see something like the following:

You entered: Boo *Hoo!*

Message: | Boo Hoo! | Submit

The browser is being given literal HTML from the Dust template, so it interprets the HTML verbatim! What do you suppose would happen if, instead of <em>, an attacker used <script> tags? Yes, you guessed it. They could get the browser to execute arbitrary JavaScript! Let's try it.

Creating an HTML attack page

All our attacker needs to do now is to get a potential victim to click on a link that looks something like `http://localhost:8000/echo?message=Hi<script src='http://localhost:8000/js/evil.js'></script>`. As you might expect, once the victim clicks on the link, the browser will see the <script> element, then load and execute the script.

18. Create a standalone HTML page at lusca-test/puppies.html, demonstrating a way an attacker might convince a victim to click on the link. Make its content the following:

```
<html>

<body>

      <h2>Click <a href="http://localhost:8000/echo?message=Hi&lt;script
src=&#39;http://localhost:8000/js/evil.js&#39;&gt;&lt;/script&gt;">here</a>
if you LOVE PUPPIES!</h2>

</body>

</html>
```

⚠️ *While this example uses* `localhost:8000` *for both the target of the attack and the attacker's JavaScript file, a real example would tell the browser to load the script from some other location, like* `http://evil.com/js/evil.js.`

Any naïve victims who love puppies will certainly click the link.

Let's create an evil JavaScript page that will simply put up an alert indicating that the target site has been hacked.

19. Create the file `lusca-test/public/js/evil.js` and make its content the following:
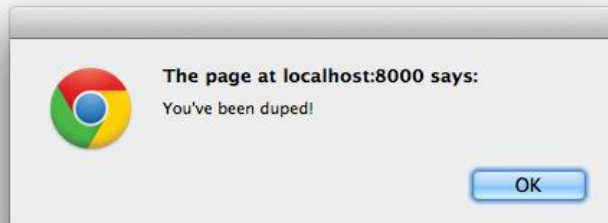
```
alert("You've been duped!");
```

View the attack result

20. Restart the web server, then open the file `lusca-test/puppies.html` in a browser. It should look like the following:

Click here if you LOVE PUPPIES!

Go ahead & click the link, since you are clearly naïve and love puppies so much. You should see the results of our evil JavaScript file's execution:

As you can see, the browser received literal HTML and simply executed it!

Close the XSS hole

Now, let's fix our site to no longer be susceptible to the XSS vulnerability.

21. Change the file `lusca-test/locales/US/en/echo.properties` to force HTML escaping:

```
greeting=You entered: {message|h}
```

22. Restart the server, reopen `lusca-test/puppies.html` in a browser, and click on the link again. This time, we should see something similar to this:

You entered: Hi<script src='http://localhost:8000/js/evil.js'></script>

Message: | Hi<script src='http://localhost:8000/js/evil.js'></script> | Submit

Now, the browser is no longer executing the HTML, because our Dust template is ensuring that it is HTML encoding any characters in the `message` value that would be interpreted literally by the browser!

What's the moral of this story? *Make sure that you're escaping content that you don't want the browser to execute!*

Add a controller to test content security policies (CSP)

Next, we're going to control from which sources our web site can download JavaScript files, fonts, styles, etc.

12. Stop the web server and issue the command `yo kraken:controller csp`, which will generate a model, controller, view, and i18n bundle in their respective places.

Update the i18n CSP bundle

13. Open the i18n bundle `lusca-test/locales/US/en/cap.properties` and change its content to the following:

```
greeting=Hello, CSP!
```

Update the CSP view

14. Open the Dust template `lusca-test/public/templates/csp.dust` to contain the following:

```
{>"layouts/master" /}


{<body}

        <script type="text/javascript"
src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min.js"></script>

        <script type="text/javascript" src="js/csp-detect.js"></script>

{/body}
```

Notice that this template is including an external jQuery script and another script from the originating website, http://localhost:8000/js/csp-detect.js.

Create the CSP detection script

15. Create a new file called `lusca-test/public/js/csp-detect.js` and set its contents to the following:

```
'use strict';
```

```
function detect() {

        var not;

        if (window.$) {

                not = '';

        } else {

                not = 'NOT ';

        }

        document.write('<h1>External script ' + not +  'loaded!</h1>');

}

detect();
```

As you can see, this script will detect whether or not the jQuery script was loaded and render a message saying so.

View default CSP behavior

If the `appsec` property's `csp` property evaluates to `false`, then there is no restriction on the sources that the browser is allowed to download from.

16. Restart the web application (`npm start`) and open the URL `http://localhost:8000/csp`, which will render the view that includes jQuery and our own `csp-detect.js`. You should see something similar to the following:

External script loaded!

As you can see, the browser successfully loaded the external jQuery script as well as our own `csp-detect.js` script, which rendered the message above. When you see this message, move on to the next step.

Restrict loading to the origin server

Now, we're going to instruct the browser that we only want it to load scripts from the originating website.

17. Update the `csp` entry of the `appsec` property to contain the following content:

```
{ "policy": { "default-src": "'self'" } }
```

This causes the Lusca middleware to include the following HTTP header in all responses:

```
Content-Security-Policy: default-src 'self';
```

This HTTP header instructs the browser to only allow scripts, styles, fonts, etc. to be loaded from the originating website. Let's see this in action.

Elicit script downloading protection

18. Restart the web server now that you've made the change to CSP configuration and open `http://localhost:8000/csp` in the browser. You should see something similar to the following:

External script NOT loaded!

The inclusion of the HTTP `Content-Security-Policy` header caused the browser to skip loading jQuery from the external site!

When you see this response, you've completed this lab!