

Real-time web application with Node.js and Express.js

Real-time web application with Node.js and Express.js



Contents

Overview	5
On completion, Delegates will be able to.....	5
Prerequisites	5
1. Introduction to Node.js.....	6
1.1 What is Node.js and what is it not?	6
1.2 Node.js Features	7
1.3 Our first Node.js script: Hello World.....	7
1.4 Review	12
2. Building your Stack.....	13
2.1 Pulling in other modules	13
2.2 Building custom modules.....	14
2.3 Core Modules.....	15
2.4 Review	16
3. Building Web Applications with the Express Framework	18
3.1 Express: Installation and Basic Setup.....	18
3.2 Routing.....	19
3.3 Views and Templating.....	20
3.4 Review	24
4. Data Transfer and Storage	25
4.1 Cookies and Sessions	25
4.2 Get and Post.....	28
4.3 Review	30
5. Asynchronous programming – Callbacks	31
5.1 Callbacks.....	31
5.2 Blocking vs. non-blocking I/O.....	31
5.3 Working within the event loop	32
5.4 Event Emitters.....	33
5.4 Review	35
6. Asynchronous programming - promises and Async/await	36
Overview	36
6.1 Three techniques for asynchronous programming	36
6.2 Promises.....	39
6.3. Async/await.....	44
6.4 Technique comparison.....	47

6.4. Exercise	47
7. Node under the hood.....	48
7.1. The Building blocks	48
7.2. The Javascript engine	49
7.3. Libuv.....	51
7.4. Putting it all together	54
7.5. coding with the event loop	55
7.6. Exercise	56
8 Working with the file system	57
8.1 Sync and Async Operations.....	57
8.2 File Manipulation	58
8.3 Directory/Folder manipulations	62
8.4 Review	63
9 Data Access with MySQL.....	64
9.1 Installing MySQL Node.js Package	64
9.2 Simple Db Connection.....	65
9.3 Using Async with MySQL Queries	68
9.4 Review	70
10 Data Access with MongoDB.....	71
10.1 Installing MongoDB.....	71
10.2 Mongoose	71
10.3 CRUD Operations	73
10.4 Review.....	75
11. Object-Oriented Programming in Node.JS	76
11.1 Prototypes	76
11.2 Constructors	76
11.3 Class Methods.....	77
11.4 Inheritance.....	77
11.5 Class vs. Object Methods.....	79
11.6 Review	80
12 User Authentication with Passport.....	81
12.1 Setting up Passport.....	81
12.2 Setting up Mongoose	83
12.3 Passport Authentication	83
12.4 Review	87

13	Unit-Testing Node Applications	88
13.1	Introduction to Unit Testing	88
13.2	Testing with Mocha and Chai	89
13.3	Review	93
14	Command Line Interface.....	94
14.1	The Built-in REPL.....	94
14.2	Custom REPL.....	95
14.3	Command-Line Applications.....	96
14.4	Building Command-Line Tools	97
14.5	Review	99
15	Real-time Communication	100
15.1	Introduction to real-time applications	100
15.2	Preparing AJAX.....	101
15.3	Web Sockets	103
15.4	Review	105
16.	Microservices.....	106
16.1	Overview	106
16.2	From Monolith to Microservices.....	106
16.3	Microservices basics.....	109
16.4	Design principles	110
16.5	Cross-Service interaction.....	111
16.6	Code Example.....	113
16.7	Challenges of Microservices	114
17.	Addendum.....	116
17.1	Array forEach vs for loop.....	116
18.	Glossary	118

Overview

Node.js is a software system designed for writing highly scalable Internet applications, notably web servers. Programs are written in JavaScript, using event-driven, asynchronous I/O to minimize overhead and maximize scalability. Unlike most JavaScript programs, it is not executed in a web browser, but is instead a server-side JavaScript application.

Node.js consists of Google's V8 JavaScript engine plus several built-in libraries.

On completion, Delegates will be able to

In this seminar you'll learn the fundamentals of Node.js and how to use Node.js to build lightweight, real-time full stack web-applications with Node and the Express framework. Participants will also learn how to interface Node.js with back-end databases and discover how to use several of the leading external modules for Node.js

Prerequisites

Experience in Object-Oriented Programming (OOP) with basic knowledge of JavaScript is necessary and will not be explicitly covered in this course.

HTML and a basic understanding of HTTP are necessary for handling HTTP requests and sending HTML replies.

MySQL and relational database design is necessary for the MySQL section. Tables, fields, and SQL will be briefly introduced.

1. Introduction to Node.js

This section covers:

- What is Node.js?
- How to execute a Node.js application
- How to write a Node.js application
- How to debug a Node.js application

1.1 What is Node.js and what is it not?

Node.js is an **open-source**, cross-platform **JavaScript** runtime environment for developing a diverse variety of tools and applications. Node.js is not a JavaScript framework, but many of its modules are written in JavaScript.

Node.js is primarily used for developing web applications. It was developed to allow web-based applications to use the same language on both the client and server. Previously, web-based applications used JavaScript on the client and a different language (PHP, ASP, Ruby, etc...) on the server. Because web browsers only use JavaScript, It is not feasible to implement a different language on the client. Therefore, implementing JavaScript, which is included in all modern web browsers, on the server became the goal of Node.js.

JavaScript (standardized as ECMAScript) is the language used in Node.js. The runtime environment interprets JavaScript using Google's V8 JavaScript engine. There are many other implementations of JavaScript interpreters. Each implementation may have different language specifications.

Node.js is **object-oriented** and **event-driven**. Using objects, allows for code reuse and fast development of real-world applications. The event-driven environment provides the illusion of a multi-threaded environment while executing as a single process. An event loop polls registered events and executes functions in an asynchronous mode.

Node.js is governed by the Node.js Foundation, facilitated by the Linux Foundation's Collaborative Projects program.

1.2 Node.js Features

Node.js has an **event-driven** architecture. It is capable of asynchronous I/O. These features allow rapid scaling of applications and real-time operations.

1.3 Our first Node.js script: Hello World

Node.js is commonly used for web applications. This example is a simple console application.

```
1. console.log('Hello World')
```

When executed, this will output text to the console.

```
2. #node hello.js  
3. Hello World  
4. #
```

Hands - On

Before learning Node.js, it is necessary to ensure that you have a functioning development environment. There are Node.js integrated development environments (IDE) such as WebStorm and NodEclipse. An IDE is not necessary. Any text editor may be used to create and edit Node.js source code.

Node.js is normally used as an **interpreted language**. It is not compiled. The interpreter is a program named node. You will want to run node on the command line to see console output, though it is possible to run Node.js code through a graphical user interface (GUI). Once fully functioning, a Node.js application is normally run as a service.

Step 1:

Start a text editor or IDE.

In Windows, you can use Notepad. Be forewarned that Notepad tends to add .txt to the file name that you specify. Notepad++ is an alternative text editor with more features for developers. It does not add .txt to all file names. Do not attempt to develop code using Microsoft Word.

In OSX, Atom is a simple text editor.

In Linux/Unix, there are many text editors from the simple (Nano) to the very complex (Vim and Emacs).

Step 2:

Create a new file. Enter the code shown on line 1. Your editor will likely highlight the text using different colors. Context highlighting is simply a visual tool to help developers quickly identify syntax errors.

Step 3:

Save your file. You may save it anywhere, but you will need to know exactly where the file is located. Node.js files are normally saved with a .js extension. Examples are example.js, MyFile.js, and HelloWorld.js. Because this program simply prints Hello World, it is suggested that you name the file HelloWorld.js.

Step 4:

Open a console, also known as a command prompt. If you are using an IDE, the console should be part of the IDE. Navigate to the directory/folder that contains the file you saved in step 3. Use cd to change directory in both Windows and Linux.

Once in the proper directory, execute your Node.js program using the node interpreter. To do so, simply type: node HelloWorld.js (assuming you named your file HelloWorld.js). You should see the message Hello World. If you do not, there is a problem with your system configuration.

Common problems:

- Node.js is not installed.
- You are not in the same directory/folder as your program.
- There is a syntax error in your code.

Hello Server: Building a web server in Node.js

The following application will listen on port 8080 and respond with a simple text message.

```
5. var http = require('http');
6. http.createServer(function(request, response) {
7.   console.log('Request received');
8.   response.writeHead(200, {'Content-Type':'text/plain'});
9.   response.end('Hello World\n');
10. }).listen(8080);
11. console.log('Server running on http://127.0.0.1:8080');
```

Line 5 includes the HTTP module, which includes all necessary code for receiving **HTTP** request and sending HTTP responses.

Line 6 through 10 create the server. It is a function that is triggered on an HTTP request event.

Lines 7 and 11 are console messages, used to provide proof that the application is running.

Line 8 sends the HTTP response header. In this case, 200 indicates a successful request. The content is sent as plain text, not HTML.

Line 9 ends the response with the text: Hello World.

Line 10 instructs the server to listen on port 8080.

Execute this script and it will respond that it is listening on port 8080. To see the output of the server itself, you must open a web browser and enter the URL <http://127.0.0.1:8080>. You will see the text in the web browser and a note on the console that a page request has been received.

Debugging node applications

Node.js has a built-in **debugger**. The debugger will cause a break when invoked. The user may then use the following commands:

C: Continue

N: Next step

S: Step in

O: Step out

The following example has a debugger placed in a simple for loop.

```
12. for(var i=0; i<5; i++) {
13.   debugger;
14.   console.log(i);
15. }
16. console.log('done');
```

When running a program with debugger lines, use the debug option in the command line.

```
17. #node debug example.js
18. < debugger listening on port 5858
19. connecting... ok
20. break in example.js:1
21.   1 for(var i=0; i<5; i++) {
22.     2 debugger;
23.     3 console.log(i);
24. debug>
```

The debugger stopped on line 1 as shown. The debugger will stop on branches by default. This allows you to use REPL to replace the value of variable(s) used to control the branch. Typing c will continue to the next break.

```
25. debug> c
26. break in example.js:2
27.   1 for(var i=0; i<5; i++) {
28.     2 debugger;
29.     3 console.log(i);
30.     4 }
31. debug>
```

This break is caused by the debugger line in the source code. Entering c again will jump to the next debugger line. Because the for loop already caused one break, it will not cause another while it is still looping.

```
32. debug> c
33. < 0
34. break in example.js:2
35.   1 for(var i=0; i<5; i++) {
36.     2 debugger;
37.     3 console.log(i);
38.     4 }
39. debug>
```

We are shown the output of 0 and it breaks at the debugger line again. To step through line-by-line, use n.

```
40. debug> n
41. break in example.js:3
42.   1 for(var i=0; i<5; i++) {
43.     2 debugger;
44.     3 console.log(i);
45.     4 }
46.   5 console.log('done');
47. debug> n
48. break in example.js:1
49.   1 for(var i=0; i<5; i++) {
50.     2 debugger;
51.     3 console.log(i);
52. debug> n
53. break in example.js:2
54.   1 for(var i=0; i<5; i++) {
55.     2 debugger;
56.     3 console.log(i);
57.     4 }
58. debug> quit
59. #
```

A node inspector application is also available for most platforms that provides a GUI interface for stepping through source code line by line.

Summary

Node.js is:

- Open-source
- Interpreted (using Google's V8 JavaScript engine)
- Object-oriented
- Event-driven

Node.js scripts are developed using a text editor or IDE. The interpreter node is used to execute Node.js scripts. Node.js has a built-in debugger invoked by adding debug to the command line.

1.4 Review



Questions

1. What is the primary use of Node.js?
 2. Why was JavaScript chosen as the language for Node.js?
 3. How are Node.js applications executed?
 4. What are some features of the Node.js language?
-



Exercises

1. Create a Node.js program that prints your name to the console.
 2. The following code sets the variable date to the current date and time. What happens if you print the date variable instead of "Hello World" or your name?

```
var date = new Date()
```
 3. Create a Node.js program that listens on a specific port (such as 8080) and sends your name to the client regardless of the request made.
 4. (See exercise 2) Create a Node.js program that listens on a specific port (such as 8080) and sends the current date and time to the client regardless of the request made.
-

2. Building your Stack

Node.js is an **object-oriented** language. This allows for reuse of existing code. A collection of objects and/or functions is commonly referred to as a **module**. A set of modules required for an application is commonly referred to as a stack. This section covers inclusion and development of modules.

2.1 Pulling in other modules

A Node.js **stack** is collection of **modules** used to support an application. Modules are included using the `require` function. The following file is written as a module because it uses `module.exports`.

```
1. //File: module_example.js
2. module.exports = {
3.   sum: function(a,b) { return a+b}
4. };
```

In this example, the module is in the same directory or folder as the code, so the path `./` is used. The extension may be included using the full file name, but the `js` extension is implied.

```
5. var ex = require('./module_example');
6. var s = ex.sum(7,12);
7. console.log(s);
```

In the previous example, `ex` is used as a **namespace** for the included module. It may be included into global space, which risks a conflict between module function and variable names.

Executing this example will print the sum of 7 and 12.

```
8. #node require_example.js
9. 19
10. #
```

There are many modules available. You are not limited to the modules that you write. Node.js has a package manager used to locate and install modules. First, use the search option to locate the name of a module that you want. Then, use the install option to install the module. For example, if you wanted to install a package to calculate Levenshtein edit distance, you would first search using:

```
11. #npm search levenshtein
```

That will return multiple packages (mainly because many people use this as a homework problem in dynamic programming). One is named levenshtein-edit-distance. To install that package, enter:

```
12. #npm install levenshtein-edit-distance
```

When doing this, ensure that you do it in your project's main directory. The first time install a package with npm, it will create a directory named `node_modules` in your project directory. All modules that you install will be installed to that directory. This allows you to have a different stack for each project that you develop. It does create redundancy as the same package is installed in more than one place in the file system.

2.2 Building custom modules

Every Node.js file is treated as a separate module. This avoids variable and function name collisions when requiring multiple modules.

All variables and functions of a module are not available outside the module. Only exported variables and functions are available. Therefore, when creating a **module**, you export the functions and variables that you want to expose to the outside code.

```
13. //File: module_example2.js
14. module.exports = {
15.   ex_sum: function(a,b) { return in_sum(a,b) }
16. }
17. var in_sum = function(a,b) { return a+b };
```

In this example, `ex_sum` is exported and will be visible to any code that imports the module. The function `in_sum` is not exported. It may not be called directly from outside code, but may be called from functions within the module itself.


```
18. var ex = require('./module_example2');
19. var s = ex.ex_sum(7,12);
20. console.log(s);
21. var s = ex.in_sum(7,12);
22. console.log(s);
```

This code will fail on line 21 because `in_sum` is not exported. Notice that `ex_sum` works even though that function internally calls `in_sum`.

```
23. #node require_example2.js
24. 19
25. /node/code/require_example2.js:4
26. var s = ex.in_sum(7,12);
27.     ^
28. TypeError: ex.in_sum is not a function
```

2.3 Core Modules

There are many built-in modules that may be required in your applications. These modules may be required without a file path. For example, to require the `http` module, you simply use `require('http')`; In previous examples, core modules were included. See line 1.5.

A list of available modules may be printed using the Read-Eval-Print-Loop or `repl` module.

```
29. #node -pe "require('repl')._builtinLibs"
30. [ 'assert',
31.   'buffer',
32.   'child_process',
33.   'cluster',
34.   'crypto',
35.   'dgram',
36.   'dns',
37.   'domain',
38.   'events',
39.   'fs',
40.   'http',
41.   'https',
42.   'net',
43.   'os',
```

```
44. 'path',  
45. 'punycode',  
46. 'querystring',  
47. 'readline',  
48. 'stream',  
49. 'string_decoder',  
50. 'tls',  
51. 'tty',  
52. 'url',  
53. 'util',  
54. 'v8',  
55. 'vm',  
56. 'zlib' ]  
57. #
```

For documentation on these modules, see the Node.js documentation at <http://nodejs.org> (currently accessed by clicking on Docs and then on v#.#.# API).

Summary

Each file in Node.js is a module. Only exported variables and functions are available to code outside of a file. Modules are included using the require function.

2.4 Review



Questions

1. What is a Node.js module?
2. Can I use the same method name in more than one Node.js module?
3. When using npm to install modules, where are they installed?
4. How is a function exposed to code outside a module?
5. Assume that a function that is exported makes a call to a function that is not exported. Will an error occur if outside code executes the exported function?



Exercises

1. Write a “hello” module that exports a method named sayHello that prints Hello to the console. Use require to import the module in

another file and then calls the sayHello function. You will have two files: one for the module and one for the implementation.

2. (See exercise 1) Add a variable to the “hello” module called language. Initially set it to English. Alter the sayHello function to use if-else statements to call functions that are not exported. Implement, but do not export, each language’s function. The following example demonstrates the if-else structure of Node.js. Include any languages that you prefer.

```
if(language=='English') sayEnglish();  
else if(language=='Spanish') saySpanish();  
else if(language=='French') sayFrench();  
else console.log('Language '+language+' unknown');
```

3. Building Web Applications with the Express Framework

The primary use of Node.js is to build **web applications**. The basic functionality of a website doesn't change much from application to application. The Express framework helps developers get the basic web application in place so they can focus on the specific features of their specific application.

3.1 Express: Installation and Basic Setup

In the previous section, we created an actual application using `npm init`. For this example, a complete application is required. Create a directory for your project, init the directory, and install express as a dependency.

1. `#mkdir hello`
2. `#cd hello`
3. `#npm init`

You can set the values of the init form as you like. Once the init process is complete, you can edit the `package.json` file and alter the values.

4. `#npm install express --save`

You should now have a `node_modules` directory with `express` in it and `express` should be a dependency listed in the `package.json` file.

A **web server** must listen for incoming connections and then handle those connections. The two functions that `express` supplies are `listen` and `get`. In hypertext transfer protocol (HTTP), a request to a web server is commonly a "get" request. To create a web application, simply define the behavior of the `get` function and then start the `listen` function.

```
5. var express = require('express');
6. var app = express();
7. app.get('/',function(req,res) {
8.   res.send('Hello World');
9. });
10. app.listen(8080,function() {
11.   console.log('Listening on http://127.0.0.1:8080');
12. });
```

Lines 5 and 6 import the express framework and create a new express object. Lines 7-9 create a get request handler. The first parameter is the page. If you do not specify a page when requesting a website, the default request is '/' (also known as the index page). The actual request is ignored and the response is hard-coded to send a Hello World message to the client. Lines 10-12 start the application's listening socket on port 8080. The express framework handles all the issues of accepting connections and triggering the proper request handlers.

Start your application and access <http://127.0.0.1:8080> in a web browser. You should see your message appear. To close the application, press Ctrl-C in your console.

You should be using at least version 4 of express. You can easily see the version in your package.json file. As of this writing, version 4.14.0 is the current stable version. It is important to note the version being used. Previous to version 4, application configuration was required to set up the listening port and page routes. As of version 4, application configuration has been removed.

3.2 Routing

In express, routing refers to the path in the URL. For example, a request to access <http://www.example.com/a/web/page.html> would be the **route** /a/web/page.html as a get request (there is no indication that it would be a post request). Each route must be defined in your application.

There are two common methods for a request, **get** and **post**. The difference is that a post request has a separate set of parameters. It is common to state that only the get request has a query string. A post request can have a query string, but it isn't expected. Deciding when to use get and post is a separate discussion that deals with web design. In general, post has a cleaner appearance and allows you to send more data from the client to the server. The drawback is that clicking the "reload" or "refresh" button will usually prompt the user to resubmit all post data, which can appear frightening.

Our existing application only handles an empty get request. Let's change the response to include a link, which is another route into our application. Then, we can create a handler for that request.

```
13. app.get('/',function(req,res) {  
14.   res.send(`  
15.     <html><body>  
16.       Hello World<br>  
17.       <a href='links.html'>Links</a>  
18.     </body></html>  
19.   `);
```

```

20. });
21. app.get('/links.html',function(req,res) {
22.   res.send(`
23.     <html><body>
24.     Links:<br>
25.     <a href='http://johnbryce.co.il'>John Bryce</a>
26.     </body></html>
27.   `);
28. });
  
```

The application has been improved to send actual HTML. Because it is confusing to cram a lot of HTML into a single line, the code uses multi-line strings with the backtick.

As we continue with express, we will make more use of routes. We will be able to divide an entire website up into multiple routes such as Anonymous, Secure, and Admin. Within each route, we will be able to define multiple web pages.

3.3 Views and Templating

In the previous example, you can see that there is repetition in the HTML for each route. For a professional web page, there will be more repetition. It is common for a website to have a standard header, standard navigation menu, and standard footer. Express handles site consistency with templates.

In the previous examples, express was used simply to handle the socket messages. It does far more. In the following example, express-generator is installed as a global module and used to quickly generate a fully-functioning web application.

```

29. #npm install -g express-generator
30. /usr/bin/express → /usr/lib/node_modules/express-generator/bin/express
31. express-generator@4.14.0 /usr/lib/node_modules/express-generator
32. #cd /home/node
33. #express hello2
34. warning: the default view engine will not be jade in future
35. create : hello2
36. create : hello2/package.json
37. create : hello2/app.js
38. create : hello2/public
39. create : hello2/routes
40. create : hello2/routes/index.js
41. create : hello2/routes/users.js
42. create : hello2/views
43. create : hello2/views/index.jade
44. create : hello2/views/layout.jade
  
```

```
45. create : hello2/views/error.jade
46. create : hello2/bin
47. create : hello2/bin/www
48. create : hello2/public/javascripts
49. create : hello2/public/stylesheets
50. create : hello2/public/stylesheets/style.css
51. #
```

Notice that the express command executed on line 33 created the project folder. A common mistake is to manually create the project directory and run the command inside that directory – which creates a project inside a project. Let express create the project directory. Then, the default structure will be in place, but the required modules are not installed. Simply enter the project directory and run npm install. Then, run npm start to test the project.

```
52. #cd hello2
53. #npm install
54. body-parser : 1.15.2
55. cookie-parser : 1.4.3
56. debug : 2.2.0
57. express : 4.14.0
58. jade : 1.11.0
59. morgan : 1.7.0
60. serve-favicon : 2.3.0
61. #npm start
62. > hello2@0,0,0 start /node/hello2
63. > node ./bin/www
```

The **web server** is running. By default, express sets it up to listen on port 3000. Open a **web browser** and enter http://127.0.0.1:3000. You should see a welcome message from the express application. Press Ctrl-C in the console to stop the application.

Looking at the directory structure, you will see:

- bin : This contains the binary code that drives the website
- node_modules : This contains modules used by the project
- public : This contains files that may be accessed directly, such as images, scripts, and styles
- routes : This contains Node.js scripts for each route defined in the app.js file
- views : This contains templates for the web pages

The main page (index) is split between routes/index.js and views/index.jade. The functionality is in the js file and the content is in the jade file. Examining the jade file makes it clear that the content includes some layout as well.

```
64. extends layout
65. block content
66.   h1= title
67.   p Welcome to #{title}
```

Line 64 indicates that jade is creates pages in an object-oriented fashion. The index view is extended from the layout view. Then, the index view overrides the block content section of the layout view. It is also clear that title is being used as a variable. To rest of the layout is in the layout.jade file.

```
68. doctype html
69. html
70.   head
71.     title= title
72.     link(rel='stylesheet', href='/stylesheets/style.css')
73.   body
74.     block content
```

This makes more sense. The layout standardizes the header of the web page and includes the standard style.css file for all pages. It is important to notice that jade is using indentation to decide which HTML elements are inside other HTML elements. If you are off by just one space, the entire layout can be thrown off.

The title variable is not defined in the layout in any way. Look at the index.js file.

```
75. var express = require('express');
76. var router = express.Router();
77. /* GET home page. */
78. router.get('/',function(req,res,next) {
79.   res.render('index', { title: 'Express' });
80. });
81. module.exports = router;
```

Line 79 renders the web page, indicating the name of the view (index) and setting the variable title.

To alter the main page, we have multiple options. We can add more variables to the index.js page, but that is all we would want to do there. We can alter layout.jade, which will alter all of the web pages. For that matter, altering style.css will alter all web pages also. We can edit index.jade, which is the content specifically for the index page.

You are not required to use views for your web pages. See the users.js file. It responds to all requests with the statement “respond with a resource.”

You will want to add more pages to your website. It is a simple, but multiple step process:

1. Create a new entry in the proper route file. We want to add links off the main route (index). Edit routes/index.js and make a copy of the router.get method. Change ‘/’ in the copy to ‘/links’ to handle requests for the links page. Then, change the res.render method in the copy to use ‘links’ instead of ‘index’.
2. Create a new view file. Again, it is usually easier to copy an existing one: cp views/index.jade views/links.jade
3. jNow you can edit the new view file to create your new web page for your web application.

Summary

Express.js greatly simplifies the process of starting a standard web server project. This section also covered standard Node.js project structure. The project.json file contains meta-information about the project. The app.js file is the main executable. Imported modules are stored in the node_modules directory. The npm tool is important for initializing, importing, and managing projects.

3.4 Review



Questions

1. What is the project.json file used for?
 2. What command is used to install a module to a project and make it a dependency in the project.json file?
 3. What is a route?
 4. How many web pages can be handled by a route?
 5. Must a route render a page from a Jade view?
-
-



Exercises

1. Create a new express.js project?
 2. Exercise 1 created a new project. Edit the main page in that project.
 3. Exercise 2 created a new project. Add a completely new page to that project.
-

4. Data Transfer and Storage

The purpose of a web server is to transfer content to a client. There are multiple methods to request and send data. It is also necessary to store data for clients. Where, when, and how long to store data are important decisions.

4.1 Cookies and Sessions

HTTP is the protocol of the web. HTTP is a **stateless** protocol. There is no defined method for storing information about users, such as the logged in status of a user or what items are in a user's shopping cart. **Cookies** and **sessions** are used to maintain the state of users. Neither is perfectly secure.

Cookies are a table of a key-value pairs. **Sessions** are also a table of key-value pairs. The choice between use of cookies and sessions is based on three criteria: storage, validity, and time.

The HTTP interface is properly defined as a request-response model. It is not a demand-response model. Cookie values are stored on the client's file system. You can make a request to have a cookie stored. You cannot make a demand that a cookie be stored. If it is vital that a value is stored, sessions should be used. Sessions are stored on the server without using the HTTP interface.

Assume that a cookie value is stored. It is on the client's file system. The server has no means of validating that the value remains unchanged. Any user can open the settings in a web browser and alter cookie values. Session values are stored on the server, outside the reach of the user.

Most developers keep nearly everything in session storage (on the server). Sessions do not last forever. They are automatically flushed to keep the server clean. Long-term storage is possible with cookie values. For example, many web pages include Google Analytics, which stores a tracking cookie on the client. The tracking cookie is set to expire in 10 years (reset to another 10 years every time it is used).

Choosing between cookie and session storage is up to the developer. The general rule is to store non-secure and long-term values in cookies. Everything else is stored in sessions.

When express was used to create a basic project, it included the cookie-parser module. Therefore, it is easy to show cookies. Edit the routes/index.js script and add a line to the get method:

```
1. router.get('/',function(req,res,next) {  
2.   console.log("Cookies: ",req.cookies);  
3.   res.render('index', { title: 'Express' });  
4. });
```

Line 83 prints the request cookies to the console when the index page is requested. Start the web application and access the index page. The console will print Cookies: {}

There are no cookies stored, yet. To set a cookie, add another line:

```
5. router.get('/',function(req,res,next) {  
6.   res.cookie('test','It Works');  
7.   console.log("Cookies: ",req.cookies);  
8.   res.render('index', { title: 'Express' });  
9. });
```

Restart the application and access the index page. The console will still print Cookies: {}

It is important to understand what is happening in the HTTP protocol, step by step.

1. Client sends a request that includes the path and all cookies stored on the client
2. Server responds by setting a cookie on the client and sending the web page
3. The client stores the new cookie and displays the web page

Notice that the request contains all cookies set on the client at the time the request is made. The server sets a cookie *after* the request has been made. The client stores the new cookie and, when the next request is made, will send that cookie.

The example on line 88 does not set an expiration time. The cookie will expire when the browser is closed. To set an expiration time, simply use a third parameter. It isn't a number of seconds. It is a specific data and time for the cookie to expire. The Date method makes it easy to set a date in the distant future. Replace line 88 with:

```
10.   res.cookie('test','It Works',{ expire : new Date()+9999 });
```

This cookie will expire 9,999 days in the future. Keep in mind that a user can delete cookies before they expire. A user can create fake cookies or change the values of existing cookies. Don't depend on cookie values for security.

From the server, you can send a request to expire cookies immediately. The function `clearCookie('CookieName')` will set the expiration time to now and cause the cookie to expire.

Now that the cookie is set, using it is simple. The cookie is an object named `req.cookie` (`req` is the name of the parameter in the request handler function). To access the value of a cookie named `test`, use `req.cookie.test`.

Cookies are far more complicated than sessions because they have to be sent through HTTP request and response messages. Session values are directly available to the server. Session handling is no longer part of the express core module. Install it to your package:

```
11. #npm install express-session --save
```

Now, require it and initialize it in the `app.js` file. In the block of require functions at the beginning of the file, add:

```
12. var session = require('express-session');
```

Then, in the section that has multiple `app.use` methods, before the routes, add:

```
13. app.use(session({  
14.   secret: '123QWE',  
15.   resave: true,  
16.   saveUninitialized: true  
17. }));
```

There are three values that must be set to avoid deprecation warnings. `Secret` is a secret key used to encrypt the session data on the server (so other server users cannot read it). `Resave` will save session values even if they are already saved. `SaveUnitialized` will save null or empty session values.

With sessions installed, you can set/read session values as an object.

```
18. req.session.test = 'It Works';  
19. console.log(req.session.test);
```

It is important to note that sessions are user-based. Express handles user identity as best it can and stores a unique set of session values per user. Therefore, setting `req.session.test` for one user will not set it for another user of your application.

Now, you have two options when you want to store a value for a user. You can place the value in a cookie on the client's machine or you can store the value on the server using sessions.

4.2 Get and Post

Cookies and sessions are used to store data. It is often necessary to transfer new data from the client to the server. This is usually performed using HTML forms. A simple HTML form can have a username and a password field with a login button.

```
1. <form action="">
2.   <input type='text' name='user' value="">
3.   <input type='password' name='pass' value="">
4.   <input type='submit' value='login'>
5. </form>
```

This form will send two values to the server when submitted: `user` and `pass`. The server sees the submission as a standard HTTP request. In this particular example, line 101 does not specify the method of the request. By default, the request is a “get” request.

Get requests send data using a **query string**. A query string is a series of names and values appended to the end of a URL. Consider the following URL:

`http://example.com/index.html?name=Scott&sbp=120`

The query string is “`name=Scott&sbp=120`” - the `?` is the delimiter indicating that the query string follows. Often, the `?` is included in the query string, although it technically is not a name or value. The query string uses the `&` character to separate name/value pairs. This example has two name/value pairs. Each name/value pair is formatted with `name=value`. If a value contains a `&` or `=`, the character must be escaped. Web browsers and most programs automatically escape special characters using some form of a URL encoding function.

Express.js makes it easy to work with query string values. They are placed in the `req.query` object. The following handler prints a query string value to the console.

```
6. router.get('/',function(req,res,next) {
7.   console.log('Name is '+req.query.name);
8.   res.render('index',{title:'Express',name:req.query.name});
9. });
```

The drawback to using query strings (get method) is that the names and values are in the URL. The URL is logged repeatedly from the web browser's history to the server's logs. Having the following URL in a log is certainly a security issue:

<http://secure.net/login.html?username=scott&password=123qwe>

Using post requests allows a form to send data to the server without a query string. The URL is just the normal URL. Names and values are passed as post data in the end of the request message. Log files will store a URL without values. There is also a benefit of size. Servers are normally implemented to limit the length of a URL, which limits the amount of data that may be placed in a query string. Post data is also limited, but the limit is commonly around 20MB.

To change a form from using get to post, simply specify the method.

```
10. <form action="" method='post'>
11.   <input type='text' name='user' value=''>
12.   <input type='password' name='pass' value=''>
13.   <input type='submit' value='login'>
14. </form>
```

Because this will submit using post, it is necessary for the route handler to define a post handler. All previous examples used the router.get method. This uses a router.post method. It is possible (and common) to define both a get and post handler for the same method. Notice that post data is placed in the req.body object.

```
15. router.post('/',function(req,res,next) {
16.   console.log('Name is '+req.body.name);
17.   res.render('index',{title:'Express',name:req.body.name});
18. });
```

Technical issues are often asked. How long can a URL be? That is dependent on the server and the web browser. Both should have hard-coded limits. Around 2000 characters is a common limit. Can a post request include a query string? Yes. Both the req.query and req.body objects will contain data, but the post handler must be used because the request is a post request. What if the same name is used multiple times in a query string or post data? It is up to the browser implementation to decide to pick one name. If more than one is sent, it is up to the server to decide which to pass to the request handler. In some cases, an array of values is passed.

Summary

Sometimes client data must be stored. Cookies provide long-term storage per client, but the data is stored on the client's computer. It is not secure. The client may alter, delete, or forge cookie data. Session data is stored short-term on the server per client. It may be trusted to a greater degree because the client has no access to it. There is a minor risk of session hijacking. Storing data in permanent storage (files or database) is an option.

Client request may come via get or post. Get requests may include data in a query string. Post requests may include data in a post data set. Post requests are preferred for sending form data, but complicate reloading web pages or saving bookmarks.

4.3 Review



Questions

1. How is a cookie value set and how is a cookie value read?
 2. How long are cookies stored?
 3. Why are cookie values untrusted?
 4. How is a session value set and how is a session value read?
 5. What is a query string?
 6. What is the difference between how a get and post request sends data?
-
-



Exercises

1. Write a handler which sets a cookie value and prints the value to the console. Notice that the value doesn't appear until the page is reloaded. Why?
 2. Write a handler that sets a session value and prints the value to the console. Notice that unlike cookies, the value is immediately available.
 3. Write a function that uses a get method to send data to the server.
 4. Using exercise 3, rewrite the function to use post method. What needs to change on the server?
-

5. Asynchronous programming – Callbacks

The node interpreter executes Node.js in an asynchronous, event-driven manner. Flow control is handled using callback functions.

5.1 Callbacks

Node.js is an **event-driven** language. Flow control can be difficult. A **callback** is the standard solution for **asynchronous** code. A callback is a function passed as a parameter to another function.

```
1. var sayHello = function() { console.log('Hello') };  
2. setTimeout(sayHello, 3000);  
3. console.log('Goodbye');
```

The function sayHello is passed as a parameter to the setTimeout function. After 3000 microseconds (3 seconds), setTimeout will invoke the sayHello function. Notice that line 3 will process before the setTimeout function is finished.

In general, a callback function may be any function. In practice, Node.js callback functions are expected to have two parameters. The first parameter is reserved for an error. The second parameter is reserved for a result. There are cases where the parameters are different, but expect them to be function(error,result) unless otherwise specified.

5.2 Blocking vs. non-blocking I/O

When working with input and output (I/O) in asynchronous code, it is often necessary to ensure that data streams are complete before moving on to other code.

Blocking I/O will stop all code from processing until the data stream is complete. Some data streams, such as keyboard input, never complete and will block indefinitely. Node.js uses an event loop to continually monitor events. Blocking the event loop from processing will cause Node.js to become unresponsive.

Non-blocking I/O will allow normal asynchronous code to process. A callback is required to signal that the data stream is complete.

```
4. var fs = require('fs');
5. console.log('Step 1');
6. var data = fs.readFileSync('../data/rain.dat');
7. console.log('Step 2');
8. fs.readFile('../data/rain.dat', function(err, data) {
9.   if(err) throw err;
10.   console.log('Step 3');
11. });
12. console.log('Step 4');
```

The read function on line 3 is a blocking read. Line 4 will not be processed until the file is fully read.

Line 5 passes a callback function as the second parameter. This function is defined inside the parameter itself, but could also be a separately defined function with just the function name supplied as a parameter. When the readFile command is finished, it will execute the callback function. While that is taking place, line 9 will execute, printing Step 4 before it prints Step 3.

5.3 Working within the event loop

Event-driven languages normally function using multiple threads. Node.js uses an event loop to handle an arbitrary number of operations in a single-threaded application.

Those familiar with JavaScript in web development have already used the event loop. All of the on* events, such as onClick and onLoad are registered with callback functions in the web browser's event loop. When an event happens, it triggers the related callback function.

While the event loop allows for asynchronous functionality, it is essentially synchronous, looping through the registered events in the order they were added. The process, simplified, looks like:

1. Execute the immediate line of code being processed
2. Poll all timers, incoming connections, data handlers, etc... to register the callback functions for all that are finished
3. Execute all pending callback functions

In the `setTimer` example, line 2 sets a timer for 3000 microseconds and registers the callback function `sayHello` to the timer. After 3000 microseconds, the timer will finish and the `sayHello` callback function will be registered in the list of pending callback functions. Then, each time the interpreter processes code to execute, the `sayHello` function will be processed until the function is complete.

While the code execution appears to be asynchronous because it executes very quickly, it is actually synchronous. It steps through the event loop in a predictable order. Registered callback functions are executed in order as well. It is helpful to understand the synchronous nature of the event loop to help troubleshoot sluggish applications due to blocking callback functions.

5.4 Event Emitters

The event emitter was created to help developers interact with the event loop. It is a wrapper module that simplifies common event loop functionality.

An event emitter contains a list of registered callback function. When the event emitter is triggered, each of the callback functions will be called in the order that they were registered. You may trigger an event emitter multiple times.

The following example creates an event emitter for a doorbell and registers a callback function to print Ding Dong when the doorbell is triggered.

```
13. var events = require('events');
14. var doorbell = new events.EventEmitter();
15. var dingdong = function() {
16.   console.log('Ding Dong');
17. }
18. doorbell.on('ring', dingdong);
```

Line 13 imports the `events` module which includes the `EventEmitter` class. The `doorbell` object is an `EventEmitter`. The callback function is defined and assigned to a variable name in lines 15-17. Line 18 registers the `dingdong` function to the `doorbell` object. Notice that the callback function is registered to the `ring` event of the `doorbell` object. The `emit` function will trigger the `ring` event.

```
19. doorbell.emit('ring'); //Prints Ding Dong
```

You may register multiple callback functions to an event. The following defines a flash function and adds it to the ring event. When the event is triggered, both the dingdong and the flash function will be executed. The order is defined by the order in which the callback functions were added to the event.

```
20. var flash = function() {  
21.   console.log('Blink Blink Blink');  
22. }  
23. doorbell.on('ring', flash);  
24. doorbell.emit('ring'); //Executes dingdong and then flash
```

Summary

Node.js uses asynchronous execution to decrease delay in servers. Blocking I/O halts asynchronous execution, increasing delay. Non-blocking I/O decreases delay, but makes flow control difficult. Callback functions are commonly used to maintain flow control in asynchronous structures. Node.js uses a synchronous, polling event loop to perform asynchronous operation using a single threaded process. The event emitter may be used to insert and trigger events in the event loop.

5.4 Review



Questions

1. What is a callback function?
 2. What makes the flow of execution asynchronous?
 3. How does blocking I/O increase delay in applications?
 4. What is the Node.js event loop?
 5. Why does the order of execution in the event loop depend on the order in which functions are registered in the loop?
 6. How does a programmer register and trigger events in the event loop?
-



Exercises

1. Write a function that prints World to the console. Write a separate function that accepts a callback function and prints Hello to the console before executing the callback function. Call the function that prints Hello with the function that prints World as the callback function.
 2. Create an event emitter. In the example in this section, the emitter had one event named ring. Add two events to your event emitter. Trigger one event and then trigger the other event.
-

6. Asynchronous programming - promises and Async/await

Overview

In our real life, should we consume a slower service like laundry, we don't expect immediate answer rather to be **called back** when the goods or service are ready. Not all interactions are synchronous by nature and for the sake of efficiency we prefer not to wait idly for a response, rather process other tasks in the interim.

Also in Node.JS, many operations are slower, mostly those that require interactions with the filesystem or networks like database queries, network requests, and file access. When our code invokes these slower operations, should it wait for a response, it will wait idly for a very long time instead of processing other tasks and requests.

For this reason, Node presents code techniques for invoking code without waiting for it, processing other tasks and request and then once the slower code is completed and potentially has a response – get back and continue from the point where we stopped. Over the years few techniques for handling asynchronous code emerged, each contains its own syntax, implications and error handling style. It's mandatory for a Node.JS developer to get acquaintance with these techniques and chose the one that suits each specific scenario best.

6.1 Three techniques for asynchronous programming

In overall, Node offers 3 techniques to invoke code in an asynchronous manner:

1. Callbacks

Callbacks are the oldest method, were introduced as part of Node version 0.1 and are considered now as an obsolete technique that many prefer to avoid. They were introduced in greater details in chapter 5 – we briefly repeat the essence of using callbacks with emphasis on the reason they were replaced with newer techniques.

Callback example

```
//let's open a file and write its content to the console  
  
fs.readFile('demofile.txt', function(err, data) {  
  
    //this is the callback function  
  
    If(err)  
  
        Console.log(err)  
  
    else  
  
        console.log(data) ;  
  
    ..  
}
```

In the example above, we use Node 'fs' module which presents filesystem methods. When we call a slower operation, readFile, we need to pass-in a function that will get called when the file content is loaded. Since reading a file is considered a slow operation (I/O) the fs library offers us not to wait idly for the response rather 'leave' a function, perform other tasks meanwhile, and then function we passed-in will get called once the file content is loaded. This type of function is called – callback.

The callback hell

When using callbacks, you might come to a point where you want to execute multiple asynchronous functions sequentially, depending on the result of the previous operation. The most common workflow example using multiple asynchronous methods might be fetching and processing data from an external source like a database. A simple asynchronous nested operation using callbacks could look like this.

Callback Nesting

```
loginUser('username', 'password', function(error,user){
  if (error){
    throw error;
  }
  else{
    loadUserProduct(user, function(error, products){
      if (error){
        throw error;
      }
      else{
        addProductsToUser(user, newProduct,function(error, products){
          if (error){
            throw error;
          }
          else{
            console.log('new Product saved');
          }
        })
      }
    })
  }
})
```

The example above demonstrates how a typical callback-based code might look like – many nested and conditional statements. This type of code is hard to read and maintain and not surprisingly it was given the name “The Callback Hell”. For this reason, two other techniques were introduced and aim to structure asynchronous code better.

2. Promises introduction

The second technique is called ‘promises’ and is supported by NodeJS since version 6.12.0. Promises are considered a much cleaner technique to handle asynchronous code, especially when it comes to chaining – invoking few asynchronous functions one after the other.

3. Async-Await introduction

The latest technique, async-await, was introduced in version 7.6.0 NodeJS, and is considered to be the most efficient and readable asynchronous programming style.

6.2 Promises

Promises were created by the community as a technique to overcome the 'callback-hell' that was described above. They are now part of ECMAScript (JS standard) and can be used without importing any libraries.

Using Promises

The essence of promises is invoking asynchronous methods like any other synchronous methods, without additional parameters (i.e. callback function) and with standard error handling techniques. A promise is basically a function that is invoked and can return 2 possible results:

1. Resolved – the operation succeeded. To tap into this event and execute our follow-up code we use the event **'then'** (example below)
2. Rejected – some error occurred. To tap into this event and execute our error handling code we use the event **'catch'**

```
loadUserProducts(user).then((products) =>{  
  
    //do something with the products  
  
}).catch((error) => {  
  
    //do something with the error  
  
})
```

The Promise is invoked and handled by the `.then()` and `.catch()` statements. Using try/catch we can handle DB errors and pass it to the `.catch()` scope and handle this error. However, if the DB returns the answer (products), we are ready for next step inside the `.then` scope, which takes us to the next subject where promises are a great advantage.

Creating Promises

Often during programming, we not only consuming promises code but also creating asynchronous promises for others to use. This is how we create a promise:

```
new Promise(function (resolve, reject) {  
  //developer custom code goes here  
  //once done simply call  
  Resolve()  
  //on error simply call  
  Reject()  
});
```

Promises are declared using the default Promise constructor containing a callback function with parameters to resolve or reject. The promise creator code goes within the callback function.

Resolve is a function used to return result data and set the Promise state to fulfilled, marking the end of the Promise execution. Resolved Promises and their values can be handled using the `then()` handler accepting a callback function with a value parameter.

Reject is a function used to transport errors from the inside of the Promise block to the outside, for example, an error thrown during an asynchronous database operation will be caught by the try-catch block, rejected and handled by the `catch()` handler of the Promise. Rejecting a Promise will set its state to rejected. It is important to handle errors thrown inside of a Promise because Node.js will simply print a warning of the rejection not being handled otherwise, basically losing the error and continuing the default workflow even if a critical part of the system broke.

```
function loadUserProducts(user) {
  return new Promise((resolve, reject) => {
    try {
      // Execute asynchronous database operation
      const data = loadDataFromDb();
      resolve(data);
    } catch (dbError) {
      // Pass up thrown error to handle with Promise
      // catch handler
      reject(dbError);
    }
  });
}
```

Chaining promises

At that point you might wonder why Promises are considered superior to callbacks – using a single promise requires to pass a callback so what's the big difference? The real power of promises comes when calling few Promises one after another. The example below demonstrates performing multiple actions sequentially without nesting and callbacks in a much more readable code: user login, loading the user's products and perform some logic on the given products:

```
loginUser(username, password)

  .then((user => loadUserProduct(user))

  .then((products) =>{

    //do something with the products

  }).catch((error) => {

    //do something with the error

  })
```

As noted in the example above, multiple Promises can be executed one after the other using flat and simple syntax. This practice of executing multiple Promises sequentially, each time returning another Promise to be executed is called **chaining**.

Using chaining the code looks way more structured. However, this works fine for simple cases only, should you need to **conditionally** execute promises and **pass** data between promises – there's no escape from using again Promise callbacks with nesting:

```
loginUser(username, password)

  .then((user) => {

    If(user.country === 'USA')

      let products = loadUserProduct(user)

    else

      let products = getOtherCountriesProducts(user)

  }) //do something with the products

).catch((error) => {

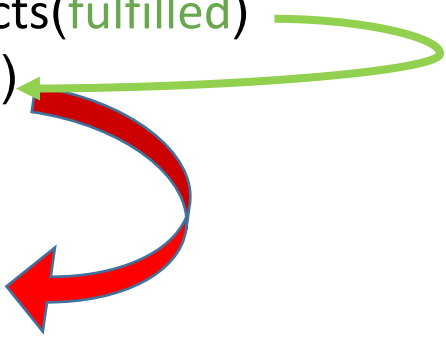
  //do something with the error

})
```

Error handling using promises

Promise allows using one of the most standards and appreciated error handling technique: try-catch. Error handling in promises is simply achieved by a single catch() clause as demonstrated in the code examples below:

```
getUserProducts(fulfilled)  
.then(rejected)  
.then()  
.then()  
.catch(error)
```



Though error handling gets much easier with Promises, a single `.catch` now catches all errors, however there are a few important things to be aware of:

1. If one promise within a chain is rejected all the following promises **won't** be called
2. If the catch clause is not present, the Node process might exit(!). Node.js will emit an event – **UnhandledRejection**. If this event is not handled Node.js will output a warning to the console and might close the process (the exact behavior is now under discussions and changes frequently). It's strongly recommended to always catch promise errors

The downside of promises

Though promises are a better technique than callbacks, there are a few drawbacks that paved the way for even better technique:

- Get lost in nesting – large and conditional code can still be too nested and not clean
- No stack information on errors – error that was thrown inside a promise and was not caught, is not presented with the right promise name and function and it is hard to understand where the error really happened
- Debugging is painful – you cannot set a breakpoint on promises that have no body for example:

```
loginUser(username, password)

.then(() => callAnotherPromise())

.then(() => callEvenAnotherOne())

).catch((error) => {

})
```

6.3. Async/await

Async/await is a new way to write asynchronous code and is supported by the runtime since Node.js 7.6. This technique addresses all the caveat that were presented related to former methods (callbacks and promises) and is considered the most concise and clean

Basics

With Async/await the code is completely flattened (no nesting), as readable as synchronizing code and all errors can be trapped using try-catch clause. The basic idea is very simple: whenever calling a promise (i.e. asynchronous function), just add the word **await** before the function call. That's it. All the rest is done similarly to synchronous programming:

Async/Await Syntax

```
// Awaiting an asynchronous function will return the
// resolved/fulfilled value or the return value itself if
// the function is not a Promise
const resolvedValue = await asynchronousFunction();

// A simple asynchronous function returning a Promise
async asynchronousFunction() { ... }
```

The 3 pillars of async/await

- Put the keyword 'await' before any call to asynchronous function
- Every asynchronous function must start with the keyword 'async'
- Asynchronous function must return a promise

Example:

```
async function loadProducts() {  
    const products = await loadProductsFromDb();  
    return products;  
}  
  
async function loadProductsFromDb() {  
    //code goes here  
}  
  
await loadProducts();
```

Handling errors

Async/await makes it finally possible to handle both synchronous and asynchronous errors with the same construct, good old try/catch. Also, due to its flat nature (no nesting), it's much easier to debug and understand the stack trace.

```
async function updateProducts(fetchedJson) {  
    try{  
        const parsedData = JSON.parse(fetchedJson)  
        const user = await User.logIn("username", "password")  
        const answer = await user.save(chosenProduct:products[0])  
    }  
    catch(error) {  
        //single catch to handle all type of errors  
    }  
}
```

In the above example, no matter where an error might occur, both within synchronous (JSON.parse) or asynchronous code (DB call) – it will get caught by the catch clause.

6.4 Technique comparison

	Callback	Promise	Async/Await
Nesting	Always	Sometime	Never
Try-Catch	No	Partially	Yes
Debugging	Painful	Even more painful	Natural
Run Multiple In Parallel	No	Yes	No
Bottom Line	Avoid	Use for parallel execution	Recommended

6.4. Exercise



The function below handles a typical user login flow. Please read the code and answer the question below

```
const result = loginUser('johnBryce', '12345')
.then((answer) => {
  //answer holds invalid JSON
  var user = JSON.parse(answer);
  if (user.isLogin){
    displayAlert(`welcome ${user.name}`);
  }
  else{
    displayAlert('wrong password')
  }
})
.catch((error) => {
  console.log('error is a ' + error);
})
```

1. What is the output of this program?
2. Which lines of code won't be executed? Why?
3. How would you fix the broken error handling?

7. Node under the hood

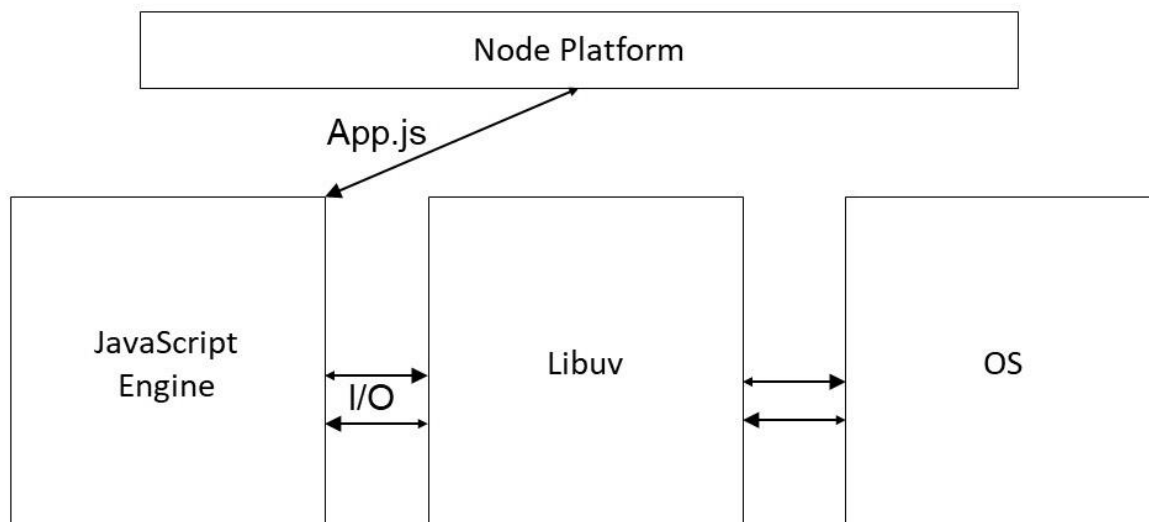
This section is about Node internals – its building blocks and how it operates internally. Understanding this can prevent critical development errors. Originally, JS was written to run inside browsers, not to serve thousands of requests on the server – therefore it utilizes a single thread only (!) which serves all the user requests, how come this model works in real-world and busy applications? What impact does it have on the way we write code?

7.1. The Building blocks

In its very essence, Node is a program that gets Javascript code as an input and executes it. However, it doesn't do all that by itself rather by orchestrating multiple open source tools that perform the heavy-lifting. Node is the glue that binds them all together.

In a nutshell, Node utilizes 3 main tools to handle a typical request that involves some IO (e.g. database call, file system access):

- **Javascript engine** – responsible to execute the code. Doesn't contain any libraries or execution model for asynchronous operations – these tasks are delegated to libuv
- **Libuv** – the heart of Node. Responsible to execute asynchronous tasks and serve back to the Javascript engine once the operation is done
- **Operating system** – being utilized by libuv. Responsible to perform low-level operations efficiently like networking and file access



7.2. The Javascript engine

To run JS code we need an engine that compiles and execute JS code. Node is using 3rd party JS engines, by default, it's packed with Google's engine which is called '**v8**'

7.2.1. Overview

The JS engine receives from Node a link to JS code and it takes care to compile, optimize and execute it. Everything related to the JS language is done by the engine. Theoretically Node can work with any JS engine (regardless of the vendor) but in practice, it uses Google's engine, v8, which is the same engine that is used by the Chrome browser.

The engine must support a well-known standard – ECMAScript – that defines the syntax and the behavior of JS. This is how Node avoids locking-in to a specific vendor and also the reason why developers can write JS that works well on multiple browsers and platforms.



NOTE: The V8 engine presents new versions frequently with new features and performance improvements

7.2.2. Scope of work

If the JS engine can execute code, why other blocks are needed? Why can't we provide it with JS code and let it perform all the work by itself? It's important where the engine falls short and other parts came in to compensate.

As long as we execute plain JS code without asynchronous and IO code, the engine can operate by itself.

Example: executed by a JS engine exclusively

```
const a = 5;
for (i=0; i<a; i++) {
    console.log(i);
}
```

JS engines have no functionality to handle asynchronous and specifically IO operations like reading from files, performing network requests and interacting with databases. Since they were designed originally to work in browsers, they are not capable of utilizing server resources for handling many IO operations. Remember that Node might also run on web server, serving thousands of requests per second, so it must take care to handle all those simultaneous IO operations very efficiently. This is where Node must onboard another tool, libuv, that takes care to handle all asynchronous operations.

Example: executed by a JS engine then delegated to libuv

```
const fs = require('fs')

//executed within the JS engine
const fileName = 'example.txt';

//IO operation - executed within libuv
fs.open(fileName, function(err, content) {

    //we got a response - excuted within the JS engine
    console.log(content)

})
```

7.2.3. How does it work?

The V8 engine, for example, is an open-source C++ library that was written by Google. It's designed to translate JS code into something that computer processor can understand – machine code (like any other programming language).

When the engine is given Javascript code, it performs the following to execute the code efficiently:

- **Parse it** – scans the code, detects errors and builds an internal representation of the code (i.e. abstract syntax tree)
- **Compile** – generates byte-code, an artifact that machine processors can execute
- **Execute** – The code is now ready to be run. V8, for example, will parse the bytecode at runtime and provide the instructions to the processors
- **Optimize** – after running once, the engine might realize that it can optimize the code. In this case, it re-compiles the code with different set of instructions that might yield faster performance

At some point, in a typical system, the code will request to perform some asynchronous operation – this is where the JS engine falls short and libuv comes into the picture.

7.3. Libuv

Libuv is an open-source framework that specializes in handling asynchronous and IO tasks efficiently, then feeding them back to the program (e.g. the JS engine). It's packed within Node program and operates under the hood

7.3.1. Overview

Libuv was written in C language (low level and very performant), it's not part of Node projects rather an independent project that Node uses.

Libuv is needed any time of the following asynchronous and IO tasks are invoked by code:

- Calls to database
- Interactions with the file-system
- Network calls
- Schedule code for later execution (i.e. timers)

© All rights reserved to John Bryce Training LTD from Matrix group

These operations are relatively slow, and the JS engine has no capabilities of executing them – this is why we need a tool that handles such tasks and bounces back to the JS engine once the task is completed

Node is often described as a ‘Single-threaded non-blocking’ language. Libuv is where this magic happens – it allows the Js engine to never block (i.w. wait) and using a single thread achieve high performance

7.3.2. Scope of work

Libuv is responsible for passing tasks between multiple tiers – The JS engine and the operation system. Should we use a restaurant analogy, libuv is not the kitchen that process tasks and good, rather the waiter that dispatches the food.

It steps in once IO related tasks are performed within the code, pulls the execution from the JS engine, approaches the operation system with a request to handle the task and dispatches back to the engine once it's ready.

It has no capabilities of running JS code (the responsibility of the JS engine), also has no capabilities of opening files or databases (the responsibility of the operation system or custom drivers) – it's the bridge that binds them together efficiently.

7.3.3. How it works

○ Starting

Libuv starts when asynchronous or IO tasks are executed by code. It takes responsibility for the task execution and promises to signal back once it's ready. It then dispatches the task to the operation system (OS) and must act thoughtfully based on the OS type. For example, different operations might get executed differently in Linux vs Windows. Libuv specializes in utilizing the OS and working with it's core libraries for IO operations.

○ Waiting for results

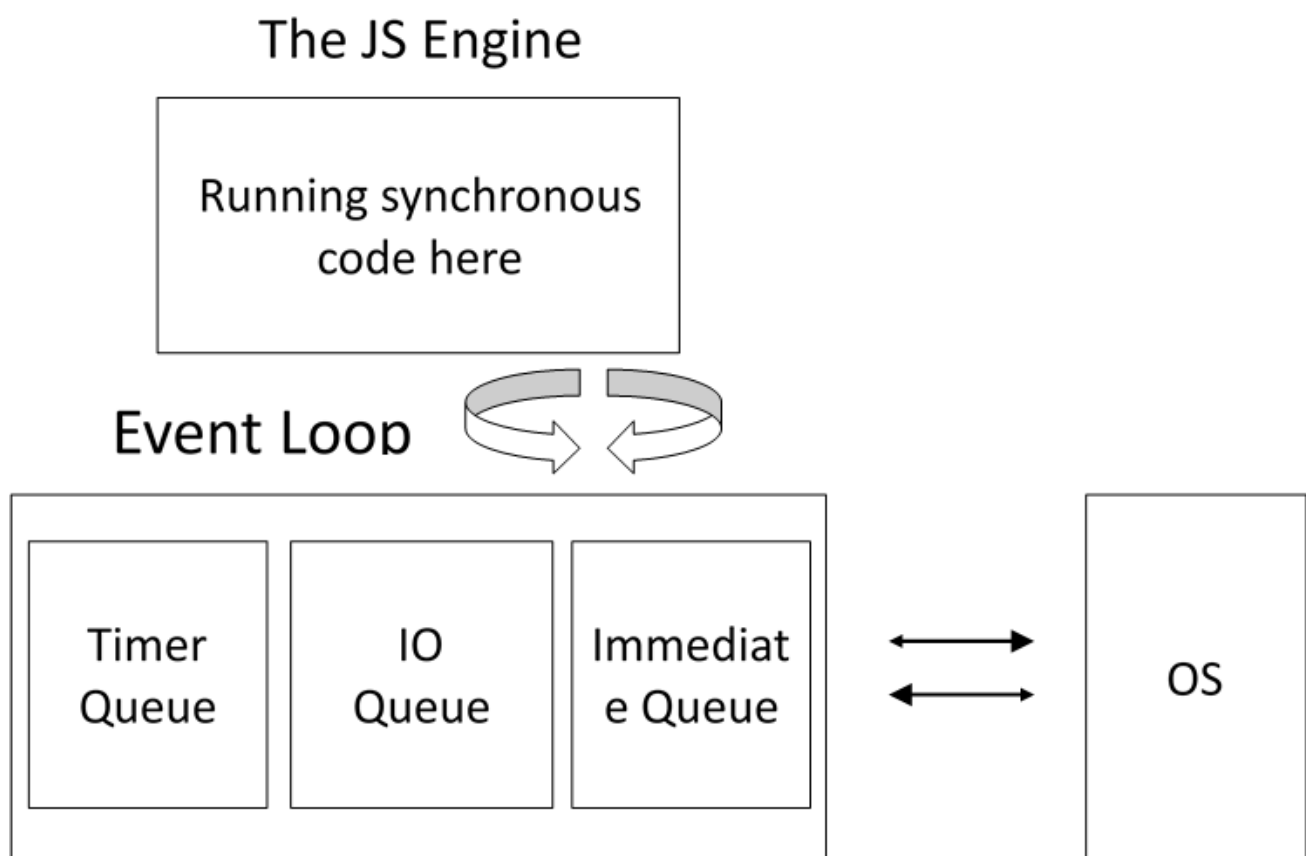
As the OS handles the task, libuv is waiting or actively polling for results. Since it must handle many tasks in-parallel it sometimes utilizes **multiple threads** (under the hood, Node is single-threaded) when approaching the OS. In other cases, it uses a single thread and leans on some advanced OS technique to get results back (epoll).

○ Dispatching the result back

As a task was handled (e.g. data came back from the database), it's about time to dispatch the result back to the JS engine so it can resume executing the code that was halted. However, since the JS engine might be busy with other code and it's single threaded – libuv had to come up with a smart mechanism for event dispatching. Libuv spread the response items (i.e. callback) across multiple queues, each has its own priority. When the response items are processed and passed to the JS engine for execution, libuv will pull items from queues by their priority - items that are stored in a prioritized queue will get handle first

These are the queues that libuv generates sorted by their priority:

- The IO queue – top priority, holds all items that are the result of I/O operations
- The timers queue – second priority, holds all timer responses (e.g. setTimeout)
- The setImmediate queue – holds all callbacks that the code asked for 'setImmediate'



7.4. Putting it all together

Now that the various parts are explained, let's see how all the pieces interact in a typical real-world scenario

1. Some **synchronous** code is executed within the JavaScript engine

```
JSON.parse(req.body)

console.log(`New request just started`)
```

2. The code approaches the **DB asynchronously**

```
DB.getData().then(() => {

    //this code is NOT executed now

    console.log(`Data arrived`)

})
```

3. Libuv handles the DB call (I/O request). Using a dedicated thread it approaches the DB and waits for result
4. The JS engine can handle other requests
5. Response arrives from DB, Libuv puts the callback code in the IO queue
6. The JavaScript engine is now free to accept a new task from the queue

```
DB.getData().then(() => {

    //this code is executed now

    console.log(`Data arrived`)

})
```


7.5. coding with the event loop

Node presents few methods for executing 'future code' using the event loop (i.e. delay code for later execution):

SetTimeout

- Allows executing callback code after specified interval

```
setTimeout( () => {  
    console.log('Show in 50 ms')  
}, 50);
```



Not recommended: won't get executed if the process dies. Use robust scheduling engines

process.nextTick

- Execute callback after the current code ends and before the next event loop task is processed

```
process.nextTick(() => {  
    console.log('Execute before others')  
})
```



Not recommended: can starve the event loop, doesn't respect other tasks

SetImmediate

- Stand on the event queue line and wait for its turn to get executed

```
setImmediate(()=>{  
    console.log('This will wait patiently')  
})
```



Recommended: use for tasks that can be executed a bit later. For example, for sending a confirmation email

7.6. Exercise



We learned about the timer and specifically about the function: `setTimeout()` which receives time and execute the code inside the scope only after the time is over.

```
Console.log('hi');  
setTimeout( () => {  
    console.log('Show after 50 ms')  
}, 50);  
console.log('bye');
```

1. Please write the program output:

1. _____
2. _____
3. _____

2. Based on what we learned, is it guarantee that the code inside the scope of the `setTimeout` function will be executed after 50 ms? Please explain why.

8 . Working with the file system

Nearly all applications have a need to work with files. JavaScript, as a web browser language, omits most file system capability. Node.js has a module, fs, that simplifies the process of working with files.

8.1 Sync and Async Operations

Synchronous operations will run to completion before another operation begins.

Asynchronous operations will start, one after the other, without waiting for each one to finish. Consider the process of reading a set of numbers from one file and saving the sum to another file. In synchronous operation, the entire set of numbers will be read before the function to sum the numbers begins. In asynchronous operation, the summing function will begin immediately after the numbers begin to load. Obviously, the summing function will not produce a sum of all the numbers.

```
1. var fs = require('fs');
2. var data = fs.readFileSync('../data/mrn.txt', 'utf8');
3. var arr = data.trim().split("\n");
4. var sum = 0;
5. arr.forEach(function(n) {
6.   sum+= parseInt(n);
7. });
8. console.log(sum);
```

Line 2 will read all of the lines of the mrn.txt file in synchronous (blocking) mode. Line 3 splits the text into an array using the newline character as a delimiter. Trim is used to avoid getting an empty element at the end of the array. Lines 5-7 sum up each value of the array. The parseInt function is used to convert the string values to integer values. Finally, the sum is printed.

Assume that this process is used in a web server and is required for every page request. Then, assume that it takes 100 microseconds to read the data file. For every page request, the entire web server will be blocked for 100 microseconds. That greatly reduces the responsiveness of the web server. The obvious solution is change the synchronous file read to an asynchronous file read.

```
9. var fs = require('fs');
10. var data = fs.readFile('./data/mrn.txt', 'utf8');
11. var arr = data.trim().split("\n");
12. var sum = 0;
13. arr.forEach(function(n) {
14.   sum+= parseInt(n);
15. });
16. console.log(sum);
```

If you were to execute this example, you will find that there is a race condition between lines 10 and 11. The trim method belongs to the built-in string class. The data variable doesn't become a string object until the readFile function finishes and returns a result. It is very likely (almost guaranteed) that line 11 will produce an error when it attempts to use the trim method of an undefined variable.

As discussed previously, callback functions are a standard solution to this problem.

```
17. var fs = require('fs');
18. fs.readFile('./data/mrn.txt', 'utf8', function(error, data) {
19.   var arr = data.trim().split("\n");
20.   var sum = 0;
21.   arr.forEach(function(n) {
22.     sum+= parseInt(n);
23.   });
24.   console.log(sum);
25. });
```

Line 18 includes a callback function that will be called when the data file is completely read. There is no race condition. Please see section 5 for examples of solving the asynchronous problem using the promise module or the async module.

8.2 File Manipulation

When working a files system, it is necessary to perform basic file manipulation. For example, it is necessary to ensure that a file exists before attempt to open it. The fs module includes many file manipulation functions.

The stat function is used to pull the status of a file or directory. As with most of Node.js, the stat function is asynchronous. A common pitfall is to execute the stat function and then attempt to immediately use the return value. The stat function will be processing when the attempt to use the return value (which will likely be undefined) executes. The following is a simple example of the stat function using a callback function.

```
26. var fs = require('fs');
27. fs.stat('./file_manipulation.js',function(err,stat) {
28.   console.log(stat);
29. });
```

The stat variable is an object containing many valuable values.

```
30. #node file_manipulation.js
31. { dev: 2081,
32.   mode: 33188,
33.   nlink: 1,
34.   uid: 1000,
35.   gid: 1000,
36.   rdev: 0,
37.   blksize: 16384,
38.   ino: 779,
39.   size: 103,
40.   blocks: 32,
41.   atime: Fri Nov 25 2016 08:22:54 GMT -0500 (EST),
42.   mtime: Fri Nov 25 2016 08:22:53 GMT -0500 (EST),
43.   ctime: Fri Nov 25 2016 08:22:53 GMT -0500 (EST),
44.   birthtime: Fri Nov 25 2016 08:22:53 GMT -0500 (EST) }
45. #
```

The stat function is often used to verify that a file exists and that it is in fact a file.

```
46. var fs = require('fs');
47. fs.stat('somefile',function(err,stat) {
48.   if(err) {
49.     console.log('File does not exist');
50.   } else {
51.     if(stat.isFile()) {
52.       console.log('It is a file');
53.     } else if(stat.isDirectory()) {
54.       console.log('It is a directory');
55.     }
56.   }
57. });
```

As with most of the file system functions, there is a synchronous option `fs.statSync` that does not use a callback function. Further, there is a simplistic function that will only indicate that a file or directory exists, but will not indicate if it is a file or directory. The following example uses the synchronous version of `exists` simply to demonstrate how the adding `Sync` changes the type of function.

```
58. if(fs.existsSync('somefile')) { console.log('It exists'); }
```

Previous sections have used `readFile` to read the contents of a file. The `writeFile` function will replace the contents of a file while `appendFile` will append content to the end of a file. Both `writeFile` and `appendFile` will create the file if it does not exist, assuming that the path does exist. For example, writing to `/path/does/not/exist/file.txt` will fail if the directory `/path/does/not/exist` does not exist. The following examples are all asynchronous, but synchronous versions exist by appending `Sync` to the function names.

```
59. //Read a file
60. fs.readFile('example.txt','utf8',function(err,data) {
61.   if(err) throw err;
62.   console.log(data);
63. });
64. //Replace contents of a file
65. fs.writeFile('example.txt','Hello',function(err) {
66.   if(err) throw err;
67. });
68. //Append to a file
69. fs.appendFile('example.txt','Hello',function(err) {
70.   if(err) throw err;
71. });
```

In these examples, the proper use of the error parameter in the callback function is shown. If there is an error, the callback function should, at a minimum, throw the error to the next higher function. Throwing the error affects the flow of the function in the same way as a return. If line 61 throws the error, line 62 will not execute.

Often, file permissions will cause errors. You will usually need to change the permission of files before executing an application. Node.js can also change the permission of files, assuming the user executing the Node.js application has permission to alter the file.

```
72. fs.chmodSync('example.txt','760');
73. fs.chownSync('example.txt',1000,500);
```

Line 72 changes the file permissions using Unix permission format. The three-digit string contains a digit for permissions user, group, and other. All Unix users are assigned to at least one user group. All files have a file owner and group owner. The permissions for user apply to the file owner. The permissions for group apply to the group owner. The permissions for other apply to everyone else.

In this example, the user has permission 7, group has permission 6, and other has permission 0. The digit is a decimal value for the binary permission string. Group's permission of 6 is equivalent to the binary 110. The three binary values are flags for read, write, and execute. Therefore, a 6 means that group users may read and write to the file, but may not execute (including rename or delete) the file.

Line 73 changes the file's owner to user ID 1000 and the file's group to group ID 500. There is no "other" ID because other refers to everyone who is not the owner of the file or in the file's group.

Renaming a file and moving a file are the same operation because the rename function allows you to include a file path in the name. If you do not include a file path, the file will simply be renamed. If you do include a file path, the file will be moved.

```
74. //Rename a file
75. fs.rename('example.txt','newname.txt',function(err) {
76.     if(err) throw err;
77. });
78. //Move a file
79. fs.rename('./example.txt','new/dir/example.txt',function(err) {
80.     if(err) throw err;
81. });
```

Finally, you can delete a file using the unlink function.

```
82. fs.unlink('example.txt',function(err) {
83.     if(err) throw err;
84. });
```

8.3 Directory/Folder manipulations

For the most part, directories (folders) are treated in the same way as files. The previous section shows examples that may be used to detect if a directory exists, change the permissions of a directory, and rename a directory. You cannot use `writeFile` or `appendFile` to create a directory. Instead, use the `mkdir` function.

```
85. var dir = './tmp';  
86. if(!fs.existsSync(dir)) { fs.mkdirSync(dir); }
```

The `unlink` function does not work with directories. Instead, use the `rmdir` function. The `rmdir` function will only work with empty directories. A common pitfall is to list the contents of a directory and assume it is empty because all files inside the directory are hidden. Both visible and hidden files must be removed before the directory may be deleted.

A common task is to work with the contents of a directory. The `readdir` function will read the contents of a directory as an array of file names.

```
87. fs.readdir('./',function(err,files) {  
88.   if(err) throw err;  
89.   files.forEach(function(filename) {  
90.     fs.stat(filename,function(err,stats) {  
91.       console.log(filename+' size is '+stats['size']);  
92.     });  
93.   });  
94. });
```

Notice how complicated the code gets as callback functions are embedded one inside the other. This example doesn't get as complicated as expected. Assume that the code needed to check the type of each file to only open files. That would be another embedded callback function. Then, assume that it would read the contents of each file. That would be another embedded callback function.

Summary

The fs module includes many functions to manipulate files and directories. Each function comes in an asynchronous version and, with the inclusion of Sync in the function name, a synchronous version. File permissions are implemented using Unix three-digit format. Most functions apply to both files and directories. Reading/writing applies to only files. Listing directory contents applies only to directories.

8.4 Review



Questions

1. File permissions are a string of three numbers. What do the numbers mean?
 2. Create a file permission that lets the file owner read and write the file, but everyone else may only read the file.
 3. Assume that there is a file function named xyzzy. What would the synchronous version of that function be?
-
-



Exercises

1. This section includes examples to list the contents of a directory, identify if a name is a file or directory, unlink files, and remove directories. Using those functions write a function that will recursively delete all the contents of a directory and then delete the directory.
 2. Using the promise or async module, rewrite lines 87 to 94 so that it avoids the embedded callback functions.
-

9. Data Access with MySQL

Many applications work with data. Often, developers will attempt to store data in text files and, in a very real sense, they try to reinvent the **database**. MySQL is a free, stable, and very popular **database engine**. There is no reason to reinvent the database when free database engines are easy to use.

Please note that the MySQL module should work seamlessly with MariaDB, a common drop-in MySQL replacement.

9.1 Installing MySQL Node.js Package

There are three ways to install the MySQL package for Node.js. Obviously, you may download the latest source code from the git repository and compile it. Only the package developers do that. If you are not compiling Node.js from source, you shouldn't attempt to compile the included packages.

Many operating systems have a package manager that includes the MySQL package. For example, Fedora includes the nodejs-mysql package in the dnf package management system. The following shows how to locate and install the package.

1. `#dnf list | grep node | grep mysql`
2. `nodejs-mysql.noarch 2.10.1-2.fc24 fedora`
3. `#sudo dnf install nodejs-mysql`

Installing the package this way has multiple advantages. It will install all dependencies. It will ensure that the package is installed properly for the operating system. It will update the package every time you use the package manager to do a system update. The drawback is that the package manager will install the package in a main library location. Instead of using `require('mysql')`, you may need to use `require('/usr/lib/node_modules/mysql')`. A solution to this problem is to set the `NODE_PATH` environment variable to look in the global module directory.

4. `#export NODE_PATH=/usr/lib/node_modules`

With `NODE_PATH` set, you can require `mysql` without the complete path.

If your operating system does not have a package manager or the package manager does not have the mysql package, you can use the Node.js package manager npm. The package is named, simply enough, mysql.

```
5. #npm install mysql
```

When installing packages, ensure that you are in the same directory as your project. This will create a directory named node_modules in your project directory and install the module to that directory. The modules in the node_modules directory are commonly referred to as the stack.

Once installed, test that the module can be included with a simple one-line script:

```
6. var mysql = require('mysql');
```

9.2 Simple Db Connection

This manual does not include how to install and set up MySQL. The following assumes that you have MySQL installed, set up, and running. All examples will assume that the username is user, the password is pass, and the database is db.

Before querying a MySQL database, it is necessary to set up a connection to the database. The connection requires the host (usually localhost, but may be another server), the user name, and a password. No database should be set up without passwords.

```
7. var mysql = require('mysql');
8. var con = mysql.createConnection({
9.   host: 'localhost',
10.   user: 'user',
11.   password: 'pass',
12.   database: 'test'
13. });
14. con.connect(function(err) {
15.   if(err) {
16.     console.log('Error connecting to DB: '+err);
17.     return;
18.   }
19.   console.log('Connected');
20. });
21. con.end();
```

Lines 8-12 set up the connection parameters. Lines 13-19 perform the connection. If there is an error, it is printed to the console. When the connection is finished, line 20 will close the database connection. Technically, the connection will close when the script ends. However, it is common that a script will finish and hang because a resource is held open. By ensuring you close the resource, you ensure that the application will run to completion.

CRUD Examples

Regardless of how data is stored, there are four common operations that we want to use: Create, Read, Update, and Delete. Each type of storage will have syntax for each of the **CRUD** operations.

To create data in MySQL, we use the insert command. First, we will create a table. When working with queries, it is common to split a string across multiple lines. In Node.js, the backtick is used to break a string across multiple lines. An added benefit to using the backtick is that you can use quotes inside the backticks without escaping them.

```
22. //Assume con is an active connection to the database
23. con.query(`
24.   create table cars(
25.     id int not null auto_increment,
26.     year int,
27.     make varchar(50),
28.     model varchar(100),
29.     primary key(id)
30.   )
31. `);
```

It is not proper to run a query and simply assume that it works. A callback function is expected with the query function. The callback function will set the error parameter if the query fails. The following example inserts a value into the cars table and checks for an error.

```
32. con.query(`
33.   insert into cars(make,year,model)
34.   values ('Chevrolet',1957,'Corvette')
35. `,function(err) {
36.   if(err) {
37.     console.log('The insert query failed! '+err);
38.     //You may want to simply kill the application at this point
39.     process.exit();
40.   }
41. });
```

After creating data, we can read data from the table. Assume that multiple records have been inserted into the cars table. The a select query will place the results of the request in the result parameter of the callback function.

```
42. con.query(`
43.   select * from cars
44. `,function(err,rows) {
45.   if(err) throw err;
46.   rows.forEach(function(row) {
47.     console.log('Model: '+row.model)
48.   }
49. });
```

When developing database driven applications, keep in mind that it is far more efficient to do data work in the database. Consider one application that loads every car from a table. Then, for each, it loads every trip for the car. Then, it sums up the miles for each trip to print out a total miles per car. There is a lot of processing involved and data transfer between the application and database. Instead, the query sent to the database should join the tables, group per car, and sum the miles all in one query.

Updates and deletes are also performed using queries. As with the create (insert) query, the only concern is an error, not a return row set. However, both update and delete do return a result that, most importantly, contains the number of rows changed. If zero rows change, the query likely failed, If every row changes, you likely messed up the query.

```
50. con.query(`
51.   update cars set year=1967 where model='Corvette'
52. `,function(err,result) {
53.   if(err) throw err;
54.   console.log('Updated '+result.changedRows+' rows');
55. });
```

9.3 Using Async with MySQL Queries

In all examples in this section, callbacks are used to avoid issues of asynchronous operation. For example, we cannot print the count of rows changed in line 52 if the query on line 45 is not complete. In section 5, the `async` module was used to simplify embedded callback functions. A series of functions is listed, the output of one feeding into the callback of the next.

Consider the following example. We want to update the miles for a trip first and then calculate the total miles per car. We do not want a race condition in which we hope the update finishes before the summary query begins (it won't). For this, embedded callback functions are used.

```
56. con.query(`
57.   insert into trips(car_id,tdate,miles)
58.   values(2,'2016-09-06',1227)
59. `,function(err) {
60.   if(err) throw err;
61.   con.query(`
62.     select make,model,sum(miles) total
63.     from cars c
64.     join trips t
65.     on c.id=t.car_id
66.     group by c.make,c.model
67.   `,function(err,rows) {
68.     if(err) throw err;
69.     rows.forEach(function(row) {
70.       console.log(row.make+' '+row.model+' total '+row.total);
71.     });
72.   });
73. });
```

It is clear here that the second query will not execute until the first query is complete. However, it looks messy. Instead, we can use the `async` module to simply list each step in order. In this example, the waterfall method is used. The waterfall method executes each function, in order, submitting the result of one to the next.

```
74. var async = require('async');
75. async.waterfall([
76.   function(callback) {
```

```

77.   con.query(`
78.       insert into trips(car_id,tdate,miles)
79.       values(2,'2016-09-06',1227)
80.   `,callback);
81. },
82. function(result,status,callback) {
83.   con.query(`
84.       select make,model,sum(miles) total
85.       from cars c
86.       join trips t
87.       on c.id=t.car_id
88.       group by c.make,c.model
89.   `,callback);
90. },
91. function(result,status,callback) {
92.   con.end()
93.   result.forEach(function(row) {
94.     console.log(row.make+' '+row.model+' total '+row.total);
95.   });
96. }
97. ],function(err,res) {
98.   if(err) console.log(err);
99. });
  
```

The order of the functions is easy to follow. They are shown clearly in order. The fact that the result is the first parameter of each function conflicts with the standard of having an error be the first parameter of each callback function.

You may have noticed that the connection is closed on line 92, not at the end of the script. If that line is moved to the end of the file, outside of the async waterfall, it will execute while async is processing the functions. A query to a closed database will be made and throw an error. If the line is omitted entirely, the script will likely hang at completion with an open resource.

Summary

MySQL is a popular, free database engine. You can install a MySQL module for Node.js using your operating system's package manager or the Node.js package manager npm. When writing multiple related queries, async may be used to ensure that the queries are performed in proper order.

9.4 Review



Questions

1. When using npm install mysql, where is the module installed?
 2. What does the acronym CRUD mean?
 3. Give an example of each part of CRUD in MySQL.
-
-



Exercises

1. Create a table in your database. Write a program to query the table and print results to the console.
 2. Write a module that has a function called getConnection. If there is no connection to the database, it will create it and return it. If there is an existing connection, it will return the existing connection.
 3. (See Exercise 2) Improve your module by adding a function called releaseConnection. Use a counter to keep track of how many functions have requested the connection. Add one for each getConnection. Subtract one for each release Connection. If, upon release, the count falls to zero, close the database connection.
-

10 . Data Access with MongoDB

Previously, MySQL was introduced as a reliable means of managing data. MongoDB is an alternative to MySQL. Classified as a **NoSQL** database engine, MongoDB is not a **relational database** like MySQL, Oracle, or MS-SQL. Instead, MongoDB is more like a document management program for a set of JSON-encoded files.

If you base your application on MySQL (or any other relational database), the data is commonly stored in the database's file space, not within your application. (It is possible, but not common, to alter the data storage path for MySQL.) MongoDB, by default, allows applications to store data in the application file space. Usually, a directory named mongo is added to the application to store the JSON files.

10.1 Installing MongoDB

To use MongoDB for a web application, you will need both the server and client executables. MongoDB is available for Windows, OS X, Linux, and Solaris. Systems that have a package manager simplify installation. The packages should be named something similar to mongodb-server and mongodb-client. In Fedora 24, installation uses the standard package manager:

```
1. #sudo dnf install mongodb-server mongodb
```

If your system does not have a package manager with MongoDB, you can download and install the latest version from <http://mongodb.com> (click the Download button in the upper right corner).

Once installed, the MongoDB server is run as a service. Whenever you have an application that will use MongoDB, you must ensure that the MongoDB server is running. Each operating system has a different method to start and stop services.

10.2 Mongoose

The mongoose module simplifies queries to a MongoDB database. The following example will create a simple MongoDB application in Node.js. First, create a directory for this application and install mongoose in it.

2. #mkdir mongo
3. #cd mongo
4. #npm install mongoose

Create a Node.js file that requires mongoose and connects to a MongoDB database.

5. var mongoose = require('mongoose');
6. mongoose.connect('mongodb://localhost/test');
7. var db = mongoose.connection;
8. db.on('error',console.log('Connection Error'));
9. db.once('open',function() {
10. console.log('Connection Works!');
11. });

Line 6 connects to the MongoDB server. Obviously, the MongoDB server must be running. You can also see that you can connect to a MongoDB server on another machine.

Line 8 sets the “on” trigger for the database. It specifically sets onError. If there is an error, it will print that there is a connection error.

Line 9 handles the open event. So far, it just prints that the connection works. This is where the database structure will be set up.

Everything in MongoDB is a schema. Schemas produce classes. From the classes, you can create an object. The following code, when placed in the open trigger for our example, will create a dogSchema, then create a dog class, and finally create an object.

12. var dogSchema = mongoose.Schema({
13. voice: String
14. });
15. dogSchema.methods.speak = function() {
16. console.log(this.voice);
17. }
18. var dog = mongoose.model('Dog',dogSchema);
19. var spot = new dog({
20. voice = 'Bark'
21. });
22. spot.speak();

Lines 12-17 set up the schema. The dog will have one variable named voice that will be a string. It will have one method named speak. Notice that variables are not given values in the schema, but functions are fully defined. You can use many different variable types: String, Number, Date, and Boolean are the most common.

Line 18 creates a dog object or model from the dog schema.

Lines 19-21 instantiate an object named spot from the dog object. The **constructor** is defined by the dogSchema. The object has the speak method. Therefore, line 22 produces the output Bark.

10.3 CRUD Operations

As with MySQL, we want to do the **CRUD** operations: create, read, update, and delete entities in our database. The previous example demonstrated how to create an object on lines 19-21. Line 16 read a value from the object (using this). Similarly, line 22 called a function on the object directly.

Updating a value for an object has two steps. The first step is to alter the object itself. Once altered, the object in your code will be changed, but the object in the database is not. To save the changes, you must call the save function, which mongoose applies to all objects.

```
23. //spot was created with voice='Bark'
24. spot.voice = 'Woof';
25. spot.speak();
26. spot.save();
```

Our database now contains an object. You can experiment and add more. Fetching objects from the database is performed using the find method. It is important to note that the find method always returns a collection, not a single object. As always, use a callback function to handle the response.

To ensure that your test is valid, you might want to copy the test used previously and delete the code that creates and saves the object. Instead, use the following code for the open trigger:

```
27. var dogSchema = mongoose.Schema({
28.   voice: String
29. });
30. dogSchema.methods.speak = function() {
31.   console.log(this.voice);
32. }
33. var dog = mongoose.model('Dog',dogSchema);
34. dog.find({},function(err,dogs) {
35.   dogs.forEach(function(dog) {
36.     dog.speak();
37.   });
```

```
38. });
```

This example fetches all dogs from the database. To limit which dogs you want, include limitations as the first parameter. The following line would find only dogs with voice equal to Bark.

```
39. dog.find({  
40.   voice: 'Bark'  
41. },function(err,dogs) {
```

Mongoose hides from the developer that it stores a unique ID for every object. The object variable is `_id`. There are multiple methods that make use of this ID. You can find by ID, find and update by ID, find and delete by ID, etc... The first trick is to get the ID of the object that you want to use.

```
42. dog.find({  
43.   voice: 'Bark'  
44. },function(err,dogs) {  
45.   dogs.forEach(function(dog) {  
46.     console.log('Dog ID: '+dog._id);  
47.   });  
48. });
```

Expect the ID to look something like 5839c9a11c29551f7a7e546a. With an ID, you can use all of the “ById” functions:

```
49. var id = '5839c9a11c29551f7a7e546a';  
50. //Finds one and only one dog  
51. dogs.findById(id,function(err,dog) {});
```

Finally, you will find it necessary to delete entities from your data store. This is done using the remove method. All mongoose objects have both a save and remove method.

```
52. spot.remove(); //Bye-bye
```

Summary

MongoDB is a NoSQL database engine. It is a good choice for storing objects. Because Node.js is an object-oriented programming language, it works well with MongoDB. Mongoose is a popular module for using MongoDB in Node.js.

10.4 Review



Questions

1. What is a NoSQL database?
 2. Why is MongoDB a good choice for storing objects in an object-oriented project?
 3. If you want to access a specific object in a MongoDB database, what property can you use?
-
-



Exercises

1. Create a MongoDB database and a schema:
`Plant({name:String,color:String});`
 2. Extend the schema from exercise 1 to a new schema Flower that includes the property `petals:Number`.
 3. Using mongoose, access the MongoDB from exercises 1 and 2. Insert a plant. Insert a flower. Notice that you cannot access the petals property using the plant object.
-

11. Object-Oriented Programming in Node.JS

Node.js is designed as an object-oriented programming language. Class inheritance is implemented using prototypes. This section covers prototypes, inheritance, and constructors.

11.1 Prototypes

Prototypes are used by Node.js to delegate **inheritance**. Every object has a property named **prototype**. When a property, such as a variable or function name, is used, the object itself is first checked. If the property exists, the object's property is used. If the property does not exist, the prototype is checked. This happens recursively.

Assume that there is an object named **foo**. The variable **foo.bar** is used. First, **foo.bar** will be checked. If that does not exist, then **foo.prototype.bar** will be checked. If that does not exist, **foo.prototype.prototype.bar** will be checked. This happens recursively until the property is located or the most parent prototype is checked.

11.2 Constructors

Every class has a **constructor** that creates an object, sets the prototype of the object, and sets this for the object. The constructor is called when the **new** keyword is used.

```
1. //Constructor
2. function Dog(name) {
3.   this.name = name;
4.   this.voice = 'Woof';
5. }
```

There is no formal class definition as is found in C++ or Java. The function itself will become a constructor when used with the **new** command. Then, as a constructor, this will be defined. For example, the following function call will not create an object.

```
6. Dog('Spot');
```

To create an object, the new keyword is used. For functionality, the object is assigned to a variable name.

```
7. var myDog = new Dog('Spot');
```

Now, MyDog is an object of the class Dog.

11.3 Class Methods

Because there is no formal class definition, class **methods** must be defined separately. While they may be attached directly to an object, they are attached to the prototype of a class.

```
8. //Class Method
9. Dog.prototype.speak = function() {
10.   console.log(this.voice);
11. }
```

The class method is used without the need of explicitly stating that it is part of the class prototype.

```
12. myDog.speak(); //Prints Woof
```

First, the property myDog.speak will be checked. It does not exist. Next myDog.prototype.speak will be checked. It does exist because myDog is an object of the Dog class and Dog.prototype.speak exists. The Dog.prototype.speak function is used.

The method uses this when referring to class variables. If voice was used instead of this.voice, the method would always produce Woof, defined in the class. Using this.voice will instead use the current object's value.

```
13. myDog.voice = 'Bark';
14. myDog.speak(); //Prints Bark
```

11.4 Inheritance

Inheritance is a key concept to object-oriented programming. Given a general class, it is easy to develop a more specific class that contains more functionality without copying all of the code from the general class. The simple act of inheritance will virtually copy all parent class functionality to a child class.

Inheritance in Node.js is merely a matter of appending to a class prototype. This can be done manually. The util module includes an inherits function that makes it easy to copy all of the variables and methods from one class to another class prototype.

```
15. function Animal() {  
16.   this.voice = 'Hello';  
17. }  
18. Animal.prototype.speak = function() {  
19.   console.log(this.voice);  
20. }
```

Similar to the Dog class used previously, this is a general Animal class with the class method speak. The util class is used to implement a Cat class with a different default voice.

```
21. function Cat() {  
22.   this.voice = 'Meow';  
23. }  
24. var util = require('util');  
25. util.inherits(Cat, Animal);
```

Now, an object of class Cat will have the speak method from the Animal class. However, the voice variable will not be overwritten with the Animal's voice. Technically, Cat.voice contains the value Meow. Cat.prototype.voice contains the value Hello. When referring to Cat.voice, the prototype will not be used because Cat.voice has a value.

```
26. var Emma = new Cat();  
27. Emma.speak(); //Prints Meow
```

While variable and method inheritance is simple using the util module, constructor inheritance is not. It is not common to have a child constructor call a parent constructor. Instead, the child constructor initializes all class variables as necessary. The following example shows how a parent constructor is called from a child class.

```
28. function Person(name) {  
29.   this.name = name;  
30. }  
31. function Employee(name,salary) {  
32.   Person.call(this,name); //Parent constructor  
33.   this.salary = salary;
```



```
34. }
35. var scott = new Employee('Scott',24200);
36. console.log('Name is '+scott.name);
```

Line 32 calls the constructor for the Parent class from inside the Employee class. The first parameter is the object itself, as this. It is followed by the required parameters for the parent's constructor.

11.5 Class vs. Object Methods

You can override a class method within a specific object by defining the function explicitly for the object. The prototype implementation will check the object first. If the method does not exist, it will then check the prototype (class) methods.

```
37. var dog1 = new Dog('Spot');
38. var dog2 = new Dog('Tinkles');
39. dog1.speak = function() {
40.   console.log('Bark Bark Woof Bark');
41. }
42. dog1.speak(); //Prints Bark Bark Woof Bark
43. dog2.speak(); //Prints Woo
```

Lines 39-41 override the speak function for the Dog class, but only for the dog1 object. The dog2 object still uses the Dog class method. Lines 42 and 43 demonstrate which method is used per object.

You can also define a method for an object that is not contained in the object's class in any way. The following code defines a fetch method for dog1. No other object of the Dog class will have this method.

```
44. dog1.fetch = function() {
45.   console.log('Fetching...');
46. }
```

Implementation of object variables is identical to implementation of object methods. Setting Dog.voice will set the voice value for all Dog objects. Setting dog1.voice will only set the voice for the dog1 object. You may create custom variables that only apply to one object.

Summary

Node.js uses prototypes to implement inheritance. With inheritance, object-oriented programming is possible. Every object has a prototype variable that is set to a parent object. Variable and method names are located by recursively searching the tree of prototype objects.

11.6 Review



Questions

1. What is inheritance?
 2. What is a constructor?
 3. How does the prototype variable implement inheritance?
 4. What is the difference between assigning a value to an object and assigning the value to a class?
-
-



Exercises

1. Create a class named Rectangle. The class should have two variables named length and width. The constructor should require both length and width. Create an object from the class.
 2. (See exercise 1) Add a method to the Rectangle class called area that returns the object's length times width.
 3. (See exercises 1 and 2) Create a class named Square that is derived from the Rectangle class. The constructor should only require one parameter for width, which is used to set both length and width of the rectangle parent class.
-

12. User Authentication with Passport

Most web-based applications have some form of **user authentication**. It could be administrators that need to login. It could be customers or clients. It could be both. Because user authentication is so common, there are multiple user authentication modules. Passport is a common user authentication module. With passport-local, you can authenticate against your local MongoDB database.

12.1 Setting up Passport

You can install the passport module to an existing express project. This example will create a new express project. First, let express create the project. Then, install passport and passport-local. Because passport uses sessions, you will want to install express-session as well.

1. #express passporttest
2. warning: the default view engine will not be jade in future
3. create : passporttest
4. ...
5. #cd passporttest
6. #npm install

The install command at the end installs all of the required modules for the application.

Inside the passporttest directory, you have two options for installing the stack of required modules. One is to edit the package.json file. The other is to manually install each module. I find it easier to manually install each module. It downloads the module and places an entry in the package.json file for you. For example, to install passport, enter the command:

7. #npm install passport --save
8. passport@0.3.2 node_modules/passport

The modules required are: express-session, passport, passport-local, mongoose, and passport-local-mongoose. Passport uses mongoose to access MongoDB and sessions to store information on the server per user.

Next, edit the app.js file to require the modules and set up passport. Your completed app.js should be very similar to the following. As versions of modules change over time, the exact code automatically created changes as well.

Add the necessary requirements for passport and mongoose. The requirements section should be:

```
9. var express = require('express');
10. var path = require('path');
11. var favicon = require('serve-favicon');
12. var logger = require('morgan');
13. var cookieParser = require('cookie-parser');
14. var bodyParser = require('body-parser');
15. var passport = require('passport');
16. var LocalStrategy = require('passport-local').Strategy;
17. var mongoose = require('mongoose');
18. var index = require('./routes/index');
19. var users = require('./routes/users');
```

You will not change the code until you get to the app.use section. Add session and passport before the final static entry:

```
20. app.use(logger('dev'));
21. app.use(bodyParser.json());
22. app.use(bodyParser.urlencoded({extended:false}));
23. app.use(cookieParser());
24. app.use(require('express-session')({
25.   secret:'123qwe',
26.   resave:false,
27.   saveUninitialized:false
28. }));
29. app.use(passport.initialize());
30. app.use(passport.session());
31. app.use(express.static(path.join(__dirname,'public')));
```

Before the catch 404 and error handler near the bottom, add code to set up passport:

```
32. //set up passport
33. var Account = require('./models/account');
34. passport.use(new LocalStrategy(Account.authenticate()));
35. passport.serializeUser(Account.serializeUser());
36. passport.deserializeUser(Account.deserializeUser());
37. mongoose.connect('mongodb://localhost/passport_local');
```

The last line starts the setup for mongoose.

12.2 Setting up Mongoose

You will likely have many objects stored in your database, not just user accounts. It is normal to separate the schema definitions and models into a separate directory to avoid cluttering app.js more than it is already cluttered. You will notice on line 33, that it assumes the account schema is defined in the models/account.js file. Create a models directory and create account.js.

```
38. var mongoose = require('mongoose');
39. var Schema = mongoose.Schema;
40. var passportLocalMongoose = require('passport-local-mongoose');
41. var Account = new Schema({
42.   username: String,
43.   password: String
44. });
45. Account.plugin(passportLocalMongoose);
46. module.exports = mongoose.model('Account',Account);
```

You have made a lot of changes to the application. It is good to test it. Run the command npm start. You should not see an error. Access <http://127.0.0.1:3000> and you should see the Express welcome page. If you do not, it is necessary to fix all errors before continuing.

12.3 Passport Authentication

Assuming everything works so far, it is time to add new content to the default “index” route. Edit the routes/index.js file. At the top, add requirements for passport and your account model.

```
47. var passport = require('passport');
48. var Account = require('../models/account');
```

Now, add a login page. Before we can login, we must have a login page. I prefer to add it below the existing router.get method, but before the module.exports = router line.

```
49. router.get('/login',function(req,res) {
50.   res.render('login',{user:req.user});
51. });
```

Notice that a new variable is being passed to the template, req.user. Add this to the index as well so it passes {title:'Express',user:req.user}. Then, you need to actually make the login page in the views directory. Create views/login.jade with:

```
52. extends layout
53.
54. block content
55.   .container
56.     h1 Login Page
57.     p.lead Please login.
58.     br
59.     form(role='form', action="/login",method="post", style='max-width: 300px;')
60.       .form-group
61.         input.form-control(type='text', name="username", placeholder='Enter Username')
62.       .form-group
63.         input.form-control(type='password', name="password", placeholder='Password')
64.       button.btn.btn-default(type='submit') Submit
65.       a(href='/')
66.         button.btn.btn-primary(type="button") Cancel
```

Pay close attention to spacing. Start the application with `npm start` and go to <http://127.0.0.1:3000/login> to ensure that the login page displays. We haven't done anything with the action of submitting the form, so that won't work. The goal is to make it appear.

Users need to know how to login. Edit the views/index.jade file and change the content to include a login button. Notice that we can use if statements in jade. This is why we pass the user variable. It allows jade to decide what to show to the user.

```
67. block content
68.   p.lead Hello World!
69.   if (!user)
70.     a(href="/login") Login
71.     br
72.     a(href="/register") Register
73.   if (user)
74.     p You are currently logged in as #{user.username}
75.     a(href="/logout") Logout
```

Before users can login, they must register an account. A registration page is helpful. That is why it was included in the new index view. First, add a handler for the register page to routes/index.js

```
76. router.get('/register',function(req,res) {
77.   res.render('register',{});
78. });
```

Then, create the view/register.jade document:

```
79. extends layout
80.
81. block content
82.   .container
83.     h1 Register Page
84.     p.lead Create an account.
85.     br
86.     form(role='form', action="/register",method="post", style='max-width: 300px;')
87.       .form-group
88.         input.form-control(type='text', name="username", placeholder='Enter Username')
89.       .form-group
90.         input.form-control(type='password', name="password", placeholder='Password')
91.       button.btn.btn-default(type='submit') Submit
92.       a(href='/')
93.         button.btn.btn-primary(type="button") Cancel
```

You may have noticed that both the register and login page use post as the method for the forms. So far, index.js only handles get requests. We know that get requests are not form submissions. For post submissions, we will use passport to create and validate accounts. Add the following handler to the routes/index.js file:

```
94. router.post('/register',function(req,res) {
95.   Account.register(new Account({username:req.body.username}),
     req.body.password,function(err,account) {
96.     if(err) {
97.       return res.render('register',{account:account});
98.     }
99.     passport.authenticate('local')(req,res,function() {
100.      res.redirect('/');
101.    });
102.  });
103. });
```

This handler will attempt to create a new account based on the submitted username and password. If it works, the user will be redirected to the main index page. Try it by going to

© All rights reserved to John Bryce Training LTD from Matrix group

your local website, clicking on the register link, and entering a new username and password. It should immediately log you in. But, was it stored in the database? Check using the command-line mongo client.

```
104. #mongo
105. MongoDB shell version 3.2.0
106. connecting to: test
107. > use passport-local
108. switched to db_passport_local
109. >db.accounts.find()
```

You should see your new account listed. The hash will be rather long (too long to print here). The username should be the username you created.

Now, we can add code to logout since creating an account logs you in immediately. Add this handler to routes/index.js:

```
110. router.get('/logout',function(req, res) {
111.   req.logout();
112.   res.redirect('/');
113. });
```

This handler does not need a view. It logs the user out and redirects to the main view (index.jade). Finally, add a post handler for login. This will trigger when a user submits the login form.

```
114. router.post('/login',passport.authenticate('local'),
115.   function(req, res) {
116.     res.redirect('/');
117.   });
```

This will attempt to log the user in. If it works, the user will be logged in and redirected to the main index page. If it fails, the user will not be logged in. Depending on the version and settings in passport, the user should see an “Unauthorized” error message.

Summary

Passport simplifies the process of handling user accounts and securing web pages. It can use local authentication with mongoose and MongoDB. It can also use many social media authentication services, such as Facebook.

12.4 Review



Questions

1. What task does the passport method simplify?
 2. Logging in and logging out are two functions provided by Passport. What is another very important function?
-
-



Exercises

1. Create a passport (and express) project.
 2. Extend exercise 1 by adding a register function so an account can be created. This requires writing a register page and register handler.
 3. Extend exercise 2 by adding a login page and login handler. Add a logoff handler.
-

13 . Unit-Testing Node Applications

Testing is an important part of professional application development. How many tests should there be? What should be tested? The general rule is to perform a unit test on each module. There are also integration and end-to-end tests, but those are beyond the scope of this guide.

Unit testing falls into two common areas of development. In **test-first programming** (TFP), a unit test is designed while programming and tests are performed before releasing code. In **test-driven design** (TDD), all documentation and tests are developed before programming begins. Then, testing is performed before releasing code. While TFP avoids most problems, TDD is the standard for professional application development.

13.1 Introduction to Unit Testing

Unit testing should be written for each module. Each test should test one and only one function. That does not mean that you cannot make multiple assertions during a test. For example, I might check that a calculation produces the correct value with a few borderline tests and an error with invalid input. However, I build the test for the one function.

Standard programming practice is to develop functions in a step-by-step order:

1. Define what the function will do
2. Define what the input for the function will be
3. Define what the output of the function will be
4. Define all tests required to identify proper handling of borderline cases and error handling
5. Write the function

Notice that the tests are defined before the function is written. Proper application development allows the programmer to develop tests before writing code as unit tests. A unit test will stub extraneous functions and hard-code input values to a function. Then, an assertion is made for the result. If the result does not match the assertion, the unit test failed.

As an example, assume that I have a function that will calculate BMI based on a person's height and weight. BMI is weight divided by height squared. I know what the function does. I know what the input values will be, numeric values for height and weight. I know that the output will be numeric. I can define that it will be rounded to two decimal places. Next, I must define the unit tests.

There are not many borderline cases with division, but there are two cases with rounding. I need to ensure that the result rounds down when the fraction is <0.005 and it must round up when the fraction is equal to 0.005 . I must find values that give me those specific results. For a value that ends with $.49999\dots$, I can take 1.49999 for weight and 10 for height, which would be $1.4999/10^2$, which is 0.0149999 . It should round down to 0.01 . Similarly, I can ensure that 1.5 and 10 round up to 0.02 .

There is the possibility of error. If height is zero, it should trigger a division by zero error. How is that handled? This is why testing is developed first. Without testing, all functions would just throw exceptions up and it would be nearly impossible to get any substantial application running. We can assume that this function is designed to return -1 when height is zero.

Now, I have three tests that include the input and the output:

```
bmi(10,1.49999) => 0.01
```

```
bmi(10,1.5) => 0.02
```

```
bmi(0,1) => -1
```

My unit testing should run every one of those test and report if any of the tests fail to produce the asserted output.

13.2 Testing with Mocha and Chai

Mocha simplifies unit testing, assuming that you have a structured development environment. Most examples in this document do not mention the development environment. For unit test, you really need an environment that contains meta information, a modules directory, and source directories. It is easy to set this up. Just create a directory for the project and, inside that directory, run `npm init`. (Note: It is common for the Node.js `init` function to fail on removable media drives. A solution is to create the folder on your hard drive first and then move it a removable media drive.)

1. `#mkdir unittesting`
2. `#cd unittesting`
3. `#npm init`
4. This utility will walk you through creating a `package.json` file.
5. It only covers the most common items,
6. and tries to guess sensible defaults.
7. See ``npm help json`` for definitive documentation on these fields
8. and exactly what they do.
9. Use ``npm install <pkg>` --save`` afterwards to install a package
10. and save it as a dependency in the `package.json` file.
11. Press `^C` at any time to quit.
12. name: (unittesting)

```

13. version: (1.0.0)
14. description: A sample unit testing application
15. entry point: (index.js)
16. test command:
17. keywords:
18. author: John Bryce
19. license: (ISC)
20. About to write to /nodejs/apps/unittesting/package.json:
21. {
22.   "name": "unittesting",
23.   "version": "1.0.0",
24.   "description": "A sample unit testing application",
25.   "main": "index.js",
26.   "scripts": {
27.     "Test": "echo \"Error: no test specified\" && exit 1"
28.   },
29.   "author": "John Bryce",
30.   "license": "ISC"
31. }
32. Is this ok? (yes) y
33. #
  
```

Once finished, your application directory will have a package.json file. Notice that the test is clearly “no test specified.” From this point, it is normal to create an app directory for the executable code and a test directory for test files.

```

34. #mkdir app
35. #mkdir test
  
```

This testing example uses the mocha module. It also uses chai, which is better version of the Node.js standard assert function. When adding these to the application, --save is used. If you check, you will see that the init function stated that you should use --save when installing modules. Doing so will include the packages in the json file as dependencies for the project.

```

36. #npm install mocha --save
37. mocha@3.2.0 node_modules/mocha
38. #npm install chai --save
39. chai@3.5.0 node_modules/chai
  
```

Now you should have a node_modules directory in your project directory. Examine the package.json file. There will be two dependencies added, one for mocha and one for chai. With mocha installed, we can do a quick test. It won’t test anything because we haven’t developed a test, but we can execute it.

```
40. #./node_modules/.bin/mocha --reporter spec
41. Warning: Could not find any test files matching patter: test
42. No test files found
```

No test files were found, but the command worked. Edit the package.json file and replace the value of “test” with “./node_modules/.bin/mocha --reporter spec” as we just typed. Now, mocha is the default unit testing program for our project.

The next step is to define the test. If you haven’t already done so, make a test and app directory in your project. Then, edit test/bmi.js

```
43. var expect = require("chai").expect;
44. var bmi = require("../app/bmi");
45. describe("BMI Calculation Function", function() {
46.   describe("BMI Calculation Tests", function() {
47.     it("Calculates with rounding", function() {
48.       var down = bmi.bmi(10,1.49999);
49.       var up = bmi.bmi(10,1.5);
50.       var zero = bmi.bmi(0,1);
51.       expect(down).to.equal(0.01);
52.       expect(up).to.equal(0.02);
53.       expect(zero).to.equal(-1);
54.     });
55.   });
56. });
```

Line 43 imports the chai module. The mocha module executes this script and does not need to be required. Line 44 imports the implementation of the function being tested.

There are three important functions being used for this test. The describe function describes a test. You can embed multiple tests inside a single test. Only one test is embedded in this example to show that it can be done. The it function defines a test to be performed. A describe function should contain at least one it function. Inside the it function, the three tests are performed, assigning the result of each test to a variable. Then, the expect function is used to compare the result of the tests to an expected value. If any test fails to meet the expectation, the assertion will fail and the test will report an error.

You can test the application now using npm test (it will fail because the implementation file is missing). We need to implement the function. Create app/bmi.js and export the bmi function. In this example, we purposely ignore the requirements for rounding and division by zero.

```
57. exports.bmi = function(height,weight) {
58.   var bmi = weight / Math.pow(height,2);
59.   return bmi;
60. };
```

The test will now complain that the first test will fail.

```
61. #npm test
62. > unittesting@1.0.0 test /bryce/unittesting
63. > mocha --reporter spec
64. BMI Calculation Function
65. BMI Calculation Tests
66.   1) Calculates with rounding
67. 0 passing (7ms)
68. 1 failing
69. 1) BMI Calculation Function
70.   AssertionError: expected 0.149999 to equal 0.01
71.     + expected - actual
72.     -0.0149999
73.     +0.01
74.     at Context.<anonymous> (test/bmi.js:9:23)
75. npm ERR! Test filed. See above for more details.
76. #
```

That is a lot of output to state that the output of the function failed a test. It produced 0.149999 when it expected 0.01. Notice that it didn't even do the other two tests. So, now we have to fix the function to pass the tests.

```
77. exports.bmi = function(height,weight) {
78.   if(height == 0) return -1;
79.   var bmi = weight / Math.pow(height,2);
80.   return Math.round(bmi*100)/100;
81. };
```

Now, running npm test will produce no errors. Our implementation of the BMI function meets all necessary unit tests. Overall, this method of test-first development helps ensure that large-scale applications do not get quickly riddled with multiple errors caused by differences in documentation and implementation.

Summary

Unit testing improves the reliability of code released from development. Test-first programming (TFP) is a good practice that ensures all released code has been tested. Test-driven development (TDD) goes one step further by developing all tests before developing code. Mocha and Chai modules simplify unit testing in Node.js projects.

13.3 Review



Questions

1. What is unit testing?
 2. What is the difference between test-first programming and test-driven development?
 3. What are expect functions used for in unit testing?
-
-



Exercises

1. Document the tests for a function that will read in a set of grades with values from 1 to 5. Values outside that range will be ignored. The function will return the average value of all values that were between 1 and 5. If all values were ignored (or no grades were sent to the function) a zero is returned.
 2. Create a project (npm init) and develop the Mocha/Chai tests for the function in exercise 1. The parameter for the function should be an array of numbers, such as [1,5,3,88,2,0].
 3. Implement the function from exercise 1 and test it using the test from exercise 2.
-

14 . Command Line Interface

Node is commonly run from the command line interface. Besides executing a script, Node.js provides an interpreter and has multiple command line options.

14.1 The Built-in REPL

If you execute node on the command line without a file name, it will launch the built-in **read-evaluate-print-loop** (REPL). It functions as a single-line JavaScript parser and is very helpful for examining how various functions behave.

The following example will set two variables, assign their sum to the first variable, and print the value of that variable. This is trivial Node.js code. Seeing it process, line by line, in the REPL helps identify exactly what is happening.

```
1. #node
2. > x=10
3. 10
4. > y=5
5. 5
6. > x=x+y
7. 15
8. > console.log(x)
9. 15
10. undefined
11. > .exit
12. #
```

A very important command to know is `.exit`. You can also type `.help` to see the other “dot” commands. If you completely forget how to exit the REPL, you can press Ctrl-C.

You can define functions in the REPL and then execute them.

```
13. #node
14. > function sayHello() { console.log('Hello'); }
15. undefined
16. > sayHello()
17. Hello
18. undefined
19. > .exit
20. #
```


Many times, you will need multiple lines to define a function. The REPL detects this and helps you see that you are using multi-line mode.

```
21. #node
22. > function sayHello() {
23. ... console.log('Hello');
24. ... }
25. undefined
26. > sayHello()
27. Hello
28. undefined
29. > .exit
30. #
```

When troubleshooting, you will likely want to load existing code and save your work. The `.load` and `.save` commands let you load and save files while in REPL.

14.2 Custom REPL

If you developed with Ruby on Rails, you likely used Rails C. It is a custom REPL interface that loads all the required libraries and sets up the environment so you can test code at the command line. The Node.js REPL is designed to be extended as well. It is a two step process. First, require the `repl` module. Second, customize the `repl.start` function.

```
31. var repl = require('repl');
32. var replServer = repl.start([
33.   prompt: 'MyREPL > '
34. ]);
```

This custom REPL does nothing more than change the prompt. Execute it with node.

```
35. #node myrepl.js
36. MyREPL > .exit
37. #
```

You can now set specific variables or functions within the custom REPL context. The following sets a custom variable and a custom function.

```
38. var repl = require('repl');
39. var replServer = repl.start([
```

```
40. prompt: 'MyREPL > '  
41. ]);  
42. replServer.context.ans = 42;  
43. replServer.context.sayHello = function() {  
44.   console.log('Hello');  
45. }
```

When using the custom REPL, I have access to both the variable and the function.

```
46. #node myrepl.js  
47. MyREPL > ans  
48. 42  
49. MyREPL > sayHello()  
50. Hello  
51. undefined  
52. MyREPL > .exit  
53. #
```

When building a custom REPL for a specific application, you will want the REPL to require all modules used in the application and then define default values and functions as necessary. Once finished, you can launch a REPL testing environment with a single command.

14.3 Command-Line Applications

You may want to develop a command-line application that is not a REPL. That requires three things: printing to the command line, receiving input from the command line, and parsing command line arguments. Based on Unix and C programming, the output stream to the command line is called Standard Out, the input stream from the command line is called Standard In, and the command line arguments are called Argument Variables.

Printing to the console is simple and has been shown repeatedly throughout this document. The `console.log` function prints a string to the console.

```
54. console.log('This is printed to the console.');
```

Reading from the command line is difficult in asynchronous programming. The `readline` module simplifies the process with the `question` method. It poses a prompt to the user and then performs a callback function with the user's response.

```
55. var readline = require('readline');
```

```
56. var cl = readline.createInterface({
57.   input: process.stdin,
58.   output: process.stdout
59. });
60. cl.question('Enter a number: ',function(answer) {
61.   console.log("You entered "+answer);
62.   cl.close();
63. });
```

Notice that it is not possible to close the readline interface until it is finished with the question-answer method. Also, the callback function has one parameter, the answer from the user.

The command line arguments are available in the variable `process.argv`. It is an array that contains the index and value of every item on the command line, including the node command itself. See the following code.

```
64. process.argv.forEach(function(value,index) {
65.   console.log(index+' ': +value);
66. });
```

When executed with command line arguments, it produces a list of the arguments.

```
67. #node argvlist.js some command-line arguments a=42
68. 0: node
69. 1: argvlist.js
70. 2: some
71. 3: command-line
72. 4: arguments
73. 5: a=42
74. #
```

14.4 Building Command-Line Tools

Node.js is not known as a common tool for building command line tools. However, assume that you are part of a team that has produced a very large and very complex set of modules for an application. It makes sense that those modules would be used in command line tools.

For this example, assume that we have multiple modules that are fundamental to our project. Each module manages a process for the application. We have used smart design and given

each module a start, stop, and status method. Now, I want a command line tool that will allow me to define a module and give it a command to start, stop, or show status.

```
75. //We expect the module name to be the third argument
76. try {
77.   var mod = require(process.argv[2]);
78. } catch(e) {
79.   console.log('Error loading '+process.argv[2]);
80.   console.log('USAGE: node manager.js module command');
81.   process.exit();
82. }
83. //We expect the command to be the fourth argument
84. if(process.argv[3] == 'start') {
85.   mod.start()
86. } else if(process.argv[3] == 'stop') {
87.   mod.stop();
88. } else if(process.argv[3] == 'status') {
89.   mod.status();
90. } else {
91.   console.log('Command must be start, stop, or status');
92. }
```

If combined with the readline.question method used previously, a complex command line tool is easy to develop.

Summary

Node.js has a built-in REPL to help with development and testing. A custom REPL may be developed for each project. With access to command line input, output, and arguments, it is possible to build complex command line tools.

14.5 Review



Questions

1. What is a REPL used for?
 2. What are two ways to preload a Node.JS REPL with variables and functions?
 3. Describe a situation that would benefit from a command-line Node.js program.
-



Exercises

1. Launch the Node.js REPL and use it to define a variable and a function.
 2. Write a command-line tool that asks for a number and prints the factors of the number.
-

15. Real-time Communication

The web is being used for far more than delivering prepackaged information. Users often expect real-time response as information changes. Stock websites are required to update stock information in real time. Social media is expected to update news feeds in real time. It is necessary to develop a means of **real-time communication** for the user.

15.1 Introduction to real-time applications

HTTP is a request-response protocol. Once the response is complete, the communication ends. While HTTP currently provides a mechanism for holding a socket open for further communication, it is not reasonable to hold open multiple sockets that may or may not result in further use. Instead, communication between the client and server requires tools to initiate a request on-demand such that a response may be sent.

Most social media websites, such as Facebook, keep a near real-time feed going while the website is open. This is done using a timer and asynchronous JavaScript and XML (**AJAX**). JavaScript embedded in the web page periodically sends a request to the server. If there is new information for the client, the server will send it as XML. The client processes the XML and alters the HTML of the web page to show updated information to the user. This avoids keeping an open socket between the client and server, but does continuously open-close-open-close a communication socket.

Real-time communication can also be from client to server. When a user fills out a form and submits information, it normally results in a completely new page load. That is not necessary. As with the previous example, AJAX may be used to send a request to the server, filled with information from the user, to update the server.

Further, there are many websites that allow users to interact with one another. In the simplest form, a website could allow multiple users to chat in real time with one other through a text interface. While it could be done with repeated page loads, the experience is far superior using real-time communication between each of the clients and the server.

Regardless of how it is done, there are two sides to the communication: the server and the client. A developer must develop code for the client to create and handle real-time requests. At the same time, the developer must develop code for the server to receive and handle real-time requests. The client code is written in JavaScript. That is why Node.js has grown in popularity. Much of the code written for the server will mimic code for the client. However, there will be two very different applications: a server and a client.

The most common failure at this point is caused by confusion about the client-server relationship. The following is a step-by-step process.

1. The client sends a request for a web page to the server.
2. The server receives the request and responds by sending a web page to the client. The web page contains HTML, CSS, and JavaScript.
3. The client processes the response, displays the web page, and begins executing the JavaScript.
4. As some point, the client's JavaScript will generate a new request to the server. This is hidden from the user.
5. The server receives the request. It appears as any HTTP request. The server responds to the request.
6. The client receives the response and processes it. It may change content on the page. It may decide to do nothing.
7. Go back to step 4 and repeat as long as the page is still open.

The goal at this point is to develop JavaScript for the client to make repeated requests to the server and, at the same time, develop responses on the server for the new requests.

15.2 Preparing AJAX

AJAX is a standard tool built into all modern web browsers for handling real-time HTTP requests. It can be complicated. For client-side code, **JQuery** greatly simplifies the code. Adding JQuery to our express project from the previous section is rather easy. Edit the views/layout.jade document and include a script command. It should look something like:

```
1. doctype html
2. html
3. head
4.   title= title
5.   link(rel='stylesheet', href='/stylesheets/style.css')
6.   script(src='http://ajax.googleapis.com/ajax/libs/jquery/
  2.0.3/jquery.min.js')
7. body
8.   block content
```

Now, we need to plan out what will be requested and what will be sent. A simple task could be requesting and sending the current disk usage. There is a very simple module for that named `diskusage`. So, use `npm` to install `diskusage`. Then, if we assume that the request will

come from the client and go to <http://127.0.0.1:3000/diskusage>, we can see that it is basically a web request. We already know how to add new responses for the main route. Edit routes/index.js and add a response for diskusage:

```
9. var diskusage=require('diskusage');
10. router.get('/diskusage', function(req,res) {
11.   diskusage.check('/', function(err,info) {
12.     res.send('Free:'+info.free+'<br>Total:'+info.total);
13.   });
14. });
```

Start the server and check the new page. You should see a response that shows the current disk usage. This changes over time, so having it constantly updated on a web page might be useful.

The next step is to write an AJAX-based script for the client. Your express setup should have a public/javascripts directory. In that directory, create a script named ajaxDiskUsage.js.

```
15. $(document).ready(function() {
16.   getDiskUsage();
17. });
18.
19. function getDiskUsage() {
20.   $.ajax({url:'/diskusage', success:function(data) {
21.     alert(data);
22.   }});
23. };
```

Line 15 is triggered when the web page is loaded and fully rendered. It calls the getDiskUsage function defined on line 19. The AJAX query begins on line 20, accessing the diskusage page and, at this point, simply spawns an alert box with the information from the server.

To make this work, this script needs to be sent to the client. Right below the inclusion of JQuery in views/layout.jade, add a new entry for this script:

```
24. script(src='/javascript/ajaxDiskUsage.js')
```

Save your work and restart the project. Access any page and you should see an alert box containing the disk usage information.

15.3 Web Sockets

Ajax can mimic real-time communication by polling the server from the client. For truly real-time communication it is necessary to open and hold open a communication connection between the client and the server. HTTP does have a method for holding open a network connection. It is rarely used. An alternative method to open and hold open a connection is web sockets.

There are two sides to web sockets, the client and the server. The client must open a socket and connect it to the server. It is technically possible for a client to open a server socket and listen for connections. Web browsers do not implement this for security reasons. In general, users do not want to accept incoming network connections simply because they opened a web page. Therefore, the server acts as the listener and the client makes the connection.

The web browser developers have all implemented web sockets in a slightly different manner, as they have done with JavaScript in general. JQuery normalizes methods for creating a web socket and connecting it to a server. Then, it normalizes the method to send messages and the trigger to handle incoming message events. The following code is for the client, written in JQuery.

```
25. $(document).ready(function() {  
26.   var ws=new WebSocket('ws://example.com:8888');  
27.   ws.onopen=function() { alert('connected'); }  
28.   $('#send').click(function() {  
29.     ws.send($('#msg').value);  
30.   });  
31.   ws.onmessage=function(msg) {  
32.     $('#chat').value+=msg+'\n';  
33.   }  
34. });
```

Line 1 defines the ready function for the document, which is triggered when the page is fully loaded. Line 2 creates a web socket and connects to the server. Lines 3 and 7 define event listeners for the socket, a connection listener and a message listener. Line 4 assigns an event listener to the send button that calls the send function on the web socket.

With a client in place, a server socket is needed. The ws module creates a web socket server that is implemented in a very similar manner to JQuery's web socket implementation. The server socket will wait for a connection event, read a message from the client, and respond to it. The read/respond sequence may be performed as long as necessary before the server closes the socket.

```
35. var WebSocketServer=require('ws').Server;  
36. var wss=new WebSocketServer({port:8888});  
37. wss.on('connection',function(ws) {  
38.   ws.on('message',function incoming(msg) {  
39.     console.log('Received:'+msg);  
40.   });  
41.   ws.send('Hello');  
42. });
```

The key to making sense of this code is recognizing that wss is the Web Socket Server. When a connection is made, it spawns ws, a Web Socket. The web socket is closed after it responds with Hello. The connection will read a message from ws and then send a response.

This form of real-time communication can be used for many purposes beyond updating content for a single user. The server can provide an interface and act as a relay for a multi-user web-based application, such as a chat engine.

Summary

Real-time communication is becoming more popular in web development. AJAX provides near real-time communication. The client polls the server by sending requests at a set interval. The server responds with short messages that the client uses to update content on the web site. Web socket provide actual real-time communications between the client and server by implementing a direct network connection that does not use HTTP requests.

15.4 Review



Questions

1. Describe a specific content item that would be improved using real-time communication.
 2. What is AJAX used for?
 3. How does AJAX mimic real-time communication?
 4. How do web sockets differ from AJAX?
 5. What are some reasons that you can't use web sockets to directly connect one client to another client?
-
-



Exercises

1. Write an AJAX server and client. The client makes an AJAX request to the url /ping. The server responds with a page containing the text Pong. The client alerts the user with the time delay between sending and receiving the message.
 2. Write a simple server and client using web sockets. The client sends Ping to the server and the server responds with Pong. The client alerts the user with the time delay between sending and receiving the message. Notice that this is (usually) shorter than using AJAX.
-

16. Microservices

16.1 Overview

Microservices is a software design style that breaks an application into multiple small independent components which are called 'Microservices'. The Microservices typically (a) runs within a dedicated process or server (b) interact with each other through the network using mechanism like HTTP requests

The main motivation for Microservices comes from the need to decrease software complexity and deploy new features faster: software designers believe that (a) small units that are stand-alone (autonomous) will yield a simpler software (b) when deploying new features – only a Microservice that was changed should be part of the deployment, so less code is updated the deployment process gets easier and faster



Note: Microservices is used these days by the world largest companies and is considered state-of-the-art design style for large applications

16.2 From Monolith to Microservices

16.1.1. Defining Monolith

A monolithic application refers to the most popular software design style where there is only one codebase and all the components are compiled together to produce a single executable that run in one process. Historically, most of the applications are being built as Monolithic due to its simplicity – only one piece of code and hardware are needed to craft the entire application.

16.1.2. Common characteristics

Monolith applications typically conform to most of the following characteristics:

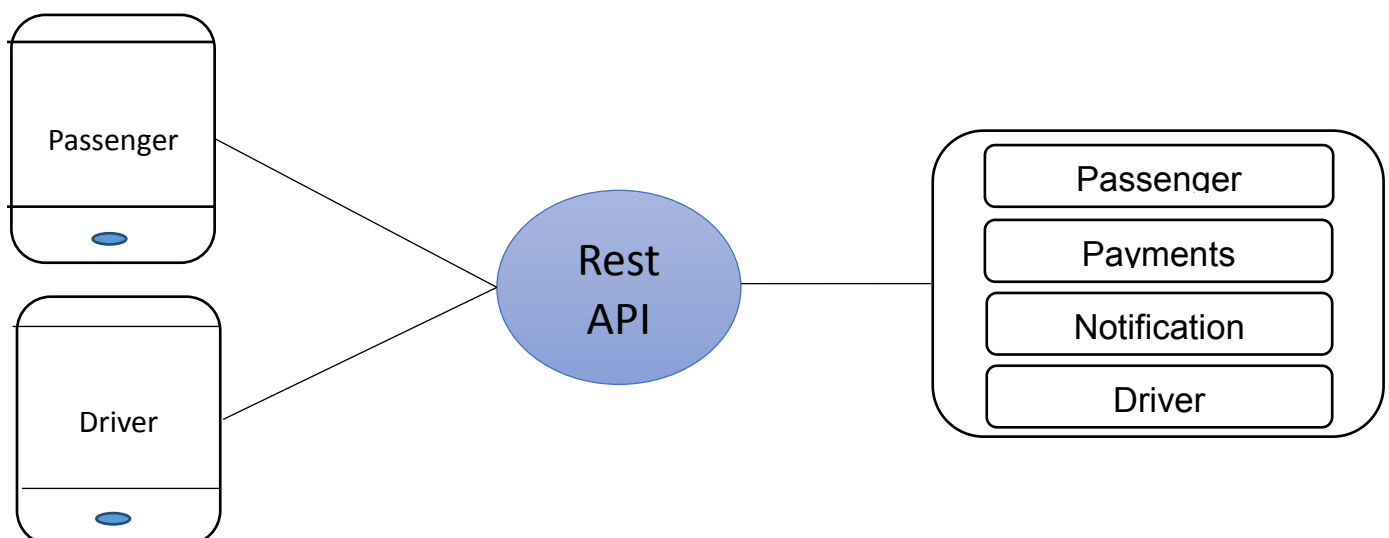
- A single repository of code

- Runs over a single process
- Every code change demand re-deploying then entire codebase
- Typically designed with a single DB that holds many tables
- Components invoke one another in-process via language-level methods
- Easiest to get up to speed fast - minimal setup

16.1.3. Example Application

The following section exemplifies a typical design of a monolith app using an imaginary application for online ordering of Taxi services - "Get Monit". This app contains 4 major modules (components): Passengers, drivers, payments, and notifications. Their code is located within the same code repository, they share many data entities and helper classes and in-production they all co-exist within the same process on the same machine. Typically, as the application grows more and more dependencies are generated between the modules. For example, (a) the payment module refers to some passenger files when it needs to approve a new payment (b) the driver module is dependent upon, and refer to some classes of the payments module when there is a need to transfer money to drivers. **In complex real-world apps, hundreds of dependencies are forming between the application components which may lead to over complicated application**

High-level view of a typical monolithic app





Note: What happens when a developer changes the code of the “Payment” module? All the dependent code, many objects from other modules, are immediately obliged to update their code. In large-scale apps, it’s easy to overlook such changes and encounter bugs

16.1.4. Challenges

As an application grows, a monolithic tends to face the following challenges:

- **Complexity** - as they grow big, many dependencies are forming between objects handling the app complexity becomes a significant challenge
- **Change fear** - every new feature demands deploying the whole system again. Consider a huge application (e.g. amazon.com) which constitutes thousands of modules – every new minor feature will demand to re-deploy the entire huge codebase
- **Limited scalability** – when a single module is very popular and generates a lot of load over the system, we can’t assign more resources or unique hardware (e.g. bigger memory/RAM, GPU card, etc.)) directly to this component rather we must provide all component with the same hardware
- **Domino effect** – a failure within a single component can take the entire application down
- **Single technology** – though different components can benefit different technologies (e.g. chat module using Node.JS, finance module using Java), in monolith apps all component run together under the same programming-language platform



Note: One other disadvantage of monoliths is concerned with embracing emerging programming frameworks/languages – since the monolith is built using a single language, one must replace the entire application to experiment new programming language. In that sense, a monolith is a ‘Catholic wedding’ with a single programming language.

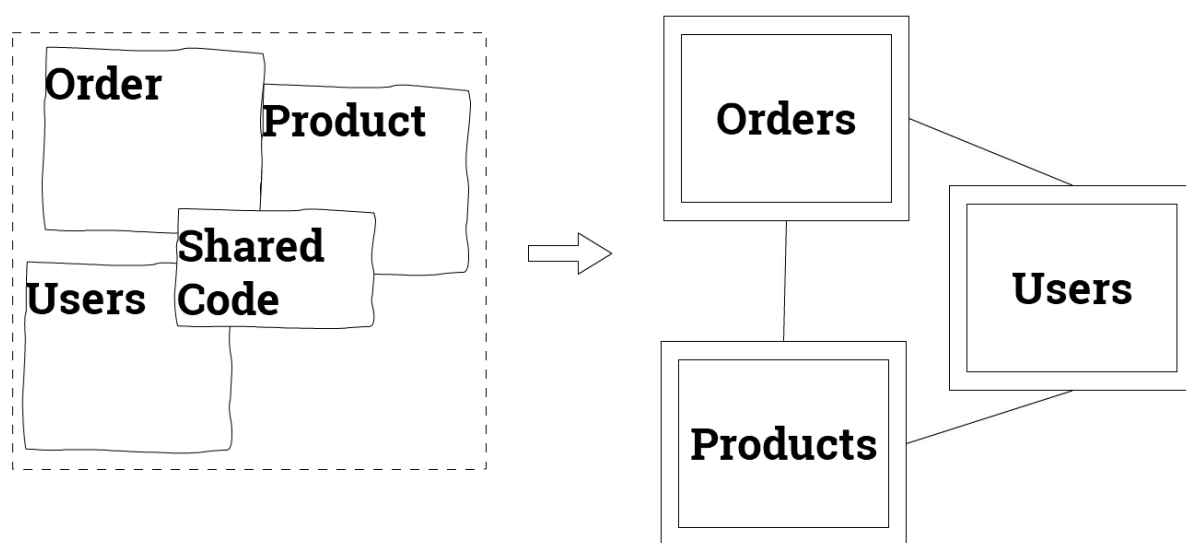
16.3 Microservices basics

16.1.1. In a nutshell

The main idea of microservices is avoiding big and complex monolith and design multiple small 'mini-software', microservices, that can live by themselves without tight dependency to any other part. For years, software architects formed many design patterns and frameworks to tame the arising complexity of big projects. Though some techniques do help, the overall result never turned big systems into simple applications that one can understand without tedious training. Microservices aims to overcome this challenge by simply avoiding big projects – if all components are small than complexity can presumably be avoided.

The Microservices paradigm is not a free lunch and introduces many new challenges like maintenance overhead due to the separation of multiple systems and the steep learning curve for developers.

From Monolith (left) to Microservices



16.1.2. Popularity and tools

In the recent years, Microservices has become the leading architecture style for big applications. Almost all the world leading sites and technology leaders embrace and use Microservices. All the cloud vendors (e.g. AWS, Google Cloud, etc.) offer engines and frameworks for deploying Microservices, there is a vibrant echo-system of products for Microservices development and deployment and all the signs suggests that it will become the standard software development style for years to come

16.4 Design principles

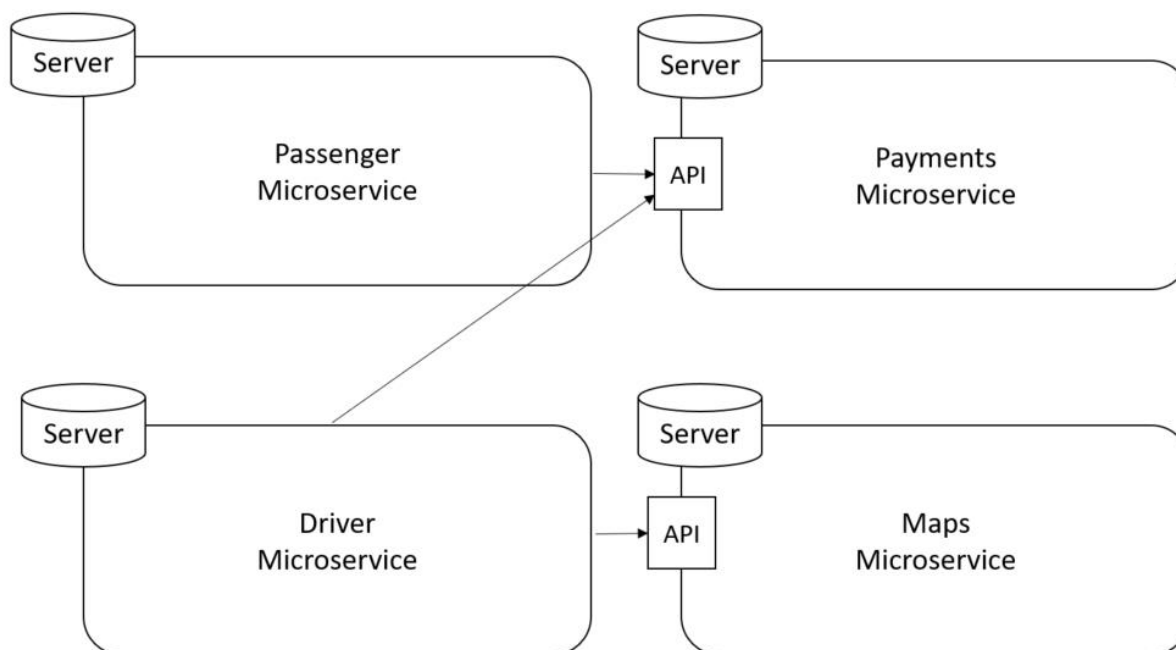
The core principle of Microservices is having small components where each (a) is small enough to avoid inner complexities (2) maximizes its uptime by serving requests even when other services are down (3) in most cases, new features require to change a single service, so the deployment process become faster and safer

To satisfy the core principles, the following guidelines and techniques are recommended when designing a Microservice based solution:

- **Small** – a service must stay relatively small so it's being developed by no more than few developers and it's rather simple to understand and change the service functionality
- **Design for failure** – if one service fails and is not available, other services should still operate and keep at least part of the systems up. This idea is also known as 'Resiliency'
- **Autonomous** – to achieve fault-tolerant systems, a service should operate over its own resources: process, DB, codebase without sharing almost anything with other services
- **The best technology for the mission** – since the services are autonomous, each Microservice can pick the best technologies that suit the business requirements best. For example, should we need to create a chat Microservice we may use Node.JS (great for hyper-networking applications) but for other business features that deal with complex statistical calculations we may use Python programming language (known for its great mathematical libraries)
- **System automation** – when breaking into multiple codebase, server and even databases, it becomes much more challenging to operate these huge stacks manually. Therefore, it's recommended to automate most of the tasks like deployment and monitoring of Microservices. Automation is mostly done with scripts and custom tools. For example, many projects include an automated deployment process which updates versions in production when a developer commits new code to source control

Example:

The following diagram shows how the 'Get Monit' application can be modeled with Microservices:



Note how each Microservice is located on his own server, each represents a business functionality (e.g. Maps feature) and the communication is done through the API of each Microservice (see next)

16.5 Cross-Service interaction

Since Microservices live on different processes or servers – they can't communicate with traditional components (Monolith) using plain functions and method calls. Typically, in a Node application, a component would use the 'require(otherComponent)' to refer to other objects and invoke methods but this technique assumes that the files are located on the same server which is obviously not the case with Microservices.

Consequently, Microservices usually interact using one or both of the following methods:

16.1.3. Request-Reply using HTTP API

Using HTTP API, a Microservice can approach other Microservice and get an immediate reply. This technique is very similar to how clients like websites, mobile applications are

approaching backend API – here caller Microservice becomes the API client. Technically this is a plain REST API that serves clients (see previous chapters about building API with Express). For example, should the Driver microservices needs to get driving instructions it will approach the Maps microservice REST API: <https://url/maps/directions/haifa/telaviv>.

This technique is very popular as its taking advantage of very popular technology that all programming languages support.

The downside of using HTTP API is the synchronous nature of this requests – the caller Microservice expects the target API to answer right away and might wait for a reply before proceeding. What if the target API is offline now or too busy? This pattern violates one of the core Microservice principles: design for failure – a Microservice should not be coupled to other services or fail when they fail.

When a Microservice must get an immediate response, and has no way to keep going without additional information – using HTTP API makes sense and is a valid choice. In other cases where the Microservice can decouple itself from the others and work synchronously – this should be the preferred method (read next).

16.1.4. Message queue

Message queue is a popular interaction pattern that provides two critical communication attributes: asynchronous interaction and delivery retry.

Should the 'Passenger' Microservice wishes to commit a payment it approaches the 'Payment' Microservice and passing the ride info, passenger ID, etc. What if the credit card company charge services are now offline – the payment Microservice won't be able to respond and the passenger will have to wait in the cab... Message queue can overcome this situation by accepting a message from the Passenger service, let it proceed (and the passenger get off the cab) and deliver the payment message to the Payment service whenever it's ready. Even after 1 hour or 1 day. This way the actual charge is processed offline. If delivering or processing fails – the message queue will try to deliver it again.

This pattern resembles a typical Post Office – a message is delivered by a postman asynchronously and if the recipient is not at home he will try to re-deliver again sometime. As opposed to HTTP interactions, message queue requires a 3rd party product or service. There are many popular message queue, most are open-source. For example, RabbitMQ, Kafka.

Also, all the cloud vendors offer their own commercial message queues.

16.6 Code Example

The following code shows how a microservices might call another microservice through HTTP request when it needs to get some information. In this example, the payments microservice is invoked by the user (e.g. Mobile application) to issue a new payment, during its execution it must ensure that the passenger indeed exists, so it approaches the Passenger microservice via HTTP. Note that the Passenger microservice might live on a different process, different machine and even on a different network.

Part 1 - The Payments microservice is invoked via HTTP post request by a mobile application

```
app.post('/api/payments', (req, res) => {

  if (!req.body.passengerId || !req.body.driverId)

    throw new Error(`Invalid input!`);

  //let's check if the passenger exists using HTTP
  //request to the passenger microservice. HTTPClient is
  //just a library that we use (out of many available
  //options) for HTTP requests

  const passengerResponse = await
  HTTPClient.get(config.url + '/' + req.body.passengerId);

  if (passengerResponse.status !== 200)

    throw new Error(`Passenger doesn't exist!`);

  //now that everything seems valid we can save new
  //payments in DB

  await DB.saveNewPayment(passengerId, driverId, amount);

  res.status(200).end()

})
```

Part2 - The Passenger microservice is invoked by the Payments microservice

```
app.get('/api/passengers/:id', (req, res)=>{

  //let's check whether the passenger exists in DB. If not
  //error status will be returned

  const existingPassenger = await
  DB.getPassenger(req.params.id);

  if(!existingPassenger)

    res.status(404).end();

  else

    res.status(200).end();
```

16.7 Challenges of Microservices

The Microservices paradigm disrupts many conventional software development patterns and presents many new challenges to developers. For example, communication between microservices is much more complex than in a monolithic application where modules invoke one another via language-level method/procedure calls. Another challenge is monitoring microservices as it's much more challenging to follow dozens of different services than a single-tier application.

More examples:

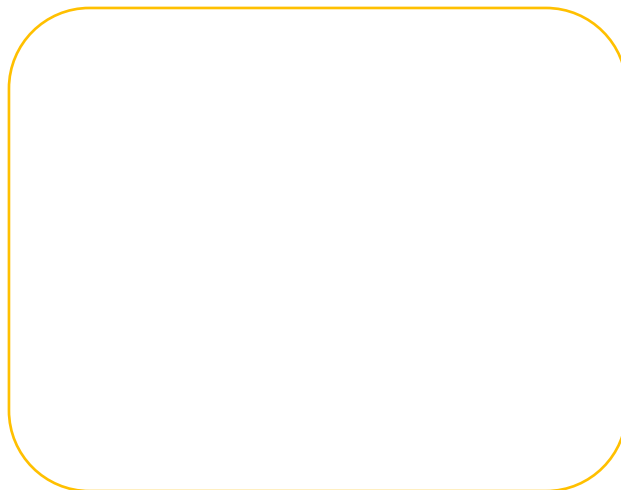
- How to track and debug interactions across the whole system?
- How to deal with versioning?
- How to deal with the partitioned database?
- How to test a service that depends on others?



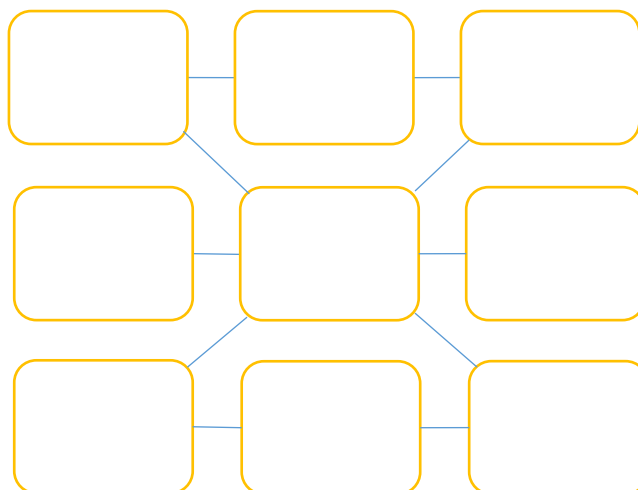
Questions

1. Which figure represents a monolithic application, and which one represents a microservice?
 2. Which figure (architecture) is easier to test?
 3. How many processes each figure has?
-

A



B



17. Addendum

This section covers programming practices that are outside the scope of this manual. As with all programming languages, Node.js programmers have formed a consensus for standards based on readability and functionality of the language.

17.1 Array `forEach` vs `for` loop

There are two common methods used to loop through an array. The `forEach` method is included with all arrays and, for each element of the array, will allow the programmer to manipulate the elements as necessary. The `for` loop is a standard programming structure that has an initializer, a comparison, and an incrementer.

In Node.js, arrays are objects. As an object, the array has predefined methods. The `forEach` method accepts a callback function. It passes each element of the array to the callback function, one at a time. The callback function may be used to work with the item from the array. Of note, the item is passed to the callback function by value, not by reference. The item in the original array will not be affected.

```
1. var arr = [10,42,12,6,7,18,8,20,9];
2. arr.forEach(function(item) {
3.   item = item+4;
4. });
5. console.log(arr);
```

The array will be printed to the console on line 5. It is clear that adding to the values did not change the original array. To alter the original values in the array, an index must be used. The second parameter of the callback function is filled with the index.

```
6. var arr = [10,42,12,6,7,18,8,20,9];
7. arr.forEach(function(item,index) {
8.   arr[index] = item+4;
9. });
10. console.log(arr);
```

In this example, the original array is altered. For some people, the use of `forEach` is easier to understand than a `for` loop. The `for` loop that performs the same function is very simple.

```
11. var arr = [10,42,12,6,7,18,8,20,9];
12. for(var index=0;index<arr.length;index++) {
13.   arr[index] = arr[index]+4;
14. }
15. console.log(arr);
```

When choosing bet

ween using `forEach` or a `for` loop, there are two issues. First, it is possible for an array in Node.js to have nonsequential indexes. If that is the case, you cannot iterate through the indexes with a `for` loop. The `forEach` method is required. Second, if you need to repeatedly iterate through a very large array, the `for` loop is significantly faster than the `forEach` method because the `for` loop does not make a copy of each element and does not make a function call. However, it is rare that a Node.js developer will work with big data, so the speed advantage of the `for` loop is minimized.

18. Glossary

AJAX: Asynchronous JavaScript HTTP request module built into most web browsers.

Asynchronous: A programming style in which operations are not initiated in a specified order and, usually, require a signal to indicate when an operation has completed.

Callback: A function passed as a parameter to a function. Primarily used to handle flow control in an asynchronous environment.

Constructor: A special method that allocates memory for an object, usually initializes properties of the object, and returns the object.

Database: A collection of data.

Database Engine: A program that manages a database.

Debugger: An application/tool used to find and fix errors in program source code.

ECMAScript: A standardized scripting programming language based on JavaScript.

Event-Driven: A programming style in which flow is determined by events rather than sequential operation of instructions.

Exports: Variables and functions that are available to code external to a module.

Get Request: The default HTTP request that places data in a query string.

HTTP: Hypertext Transfer Protocol - The protocol used for the Web.

Inheritance: A feature of object-oriented programming in which a class gains all properties and methods of another class, commonly forming a parent-child relationship.

Interpreted Language: A programming language that is run through a parser instead of being compiled to byte or machine code.

JavaScript: A collection of scripting programming languages. See ECMAScript.

JQuery: A JavaScript wrapper that normalizes the implementation of JavaScript found in popular web browsers.

Methods: Functions that are encapsulated in classes in object-oriented programming.

Module: A program designed to be included in a larger programming application.

NoSQL: A database that stores data in a way that is not a relational database, such as a collection of objects.

Object-Oriented: A programming design based on encapsulation and inheritance using classes and objects.

Open-Source: Indicates that the source code for a product will be provided and is often included with the product.

Post Request: An HTTP request that places data separate from the URL in the request package.

Prototype: A special object attached to all Node.js objects that allows for the creation of class inheritance.

Read-Eval-Print-Loop: A command-line tool that parses and executes code as it is written.

Real-Time Communication: Immediate communication between a client and server through an existing, always-on connection.

Relational Database: A tabular database that includes cross-references between tables.

Route: A logical group of web pages managed by a web server.

Stack: A collection of modules used to support a programming application.

Test-Driven-Design: Documentation and tests are developed before programming begins. Tests are performed before releasing code.

Test-First Programming: Tests are developed while programming and performed before releasing code.

Unit Test: A test of a specific function or method in a project.

User Authentication: Validating the identity of a user, usually with a user name and password.

Web Applications: An application that uses a web browser for the user interface and a web server for most functionality.

Web Browser: A program used to view a web page. Common programs are Internet Explorer, Safari, Firefox, and Chrome.

Web Server: A service that listens for HTTP requests and delivers content, usually web pages, as an HTTP result.