PART I
Internet Programming

HTTP, IP ADDRESSES, HTTP VS HTTPS,

THICK VS THIN CLIENT, GET VS POST,

SYNCHRONOUS VS ASYNCHRONOUS FUNCTIONS,

CALLBACK FUNCTIONS

HTTP BASICS

HTTP (Hypertext Transfer Protocol) is the set of rules for transferring files (text, graphic images, sound, video, and other multimedia files) on the World Wide Web. As soon as a Web user opens their Web browser, the user is indirectly making use of HTTP. HTTP is an application protocol that runs on top of the TCP/IP suite of protocols (the foundation protocols for the Internet).

Features of HTTP

- The Hypertext Transfer Protocol (HTTP) is designed to enable communications between clients and servers.
- Implements a request-response model
- Basis of all www communications
- Defines format for requests and responses
  - HTTP messages consist of requests from client to server and responses from server to client.
- Stateless protocol/service (HTTP based on stateless protocol –server does not remember the client, or what they wanted in the past.)
- All information required to service the request is sent with the request
- Open plain-text protocol
  - A text-based protocol or plain text protocol is a communications protocol whose content representation is in human-readable format.
- HTTP works as a request-response protocol between a client and server.

IP ADDRESSES

IP address is a unique string of numbers separated by periods that identifies each computer using the Internet Protocol to communicate over a network.

- IP Addresses are 4 byte numbers –based on current IPv4 standard (1984)
  - TCP/IP uses 32 bits addressing. One computer byte is 8 bits. So TCP/IP uses 4 computer bytes.
- TCP/IP address is four numbers between 0 and 255
  - because a computer byte can contain 256 different values:
- Newer IPv6 standard promises much bigger address space
- 192.168.35.211 is an example of IP address.

Domain names

A name is much easier to remember than a 12 digit number. Names used for TCP/IP addresses are called domain names. Carleton.ca is a domain name.

- Last domain specifies the type of organization or country (e.g. .com, .ca, .za).

- DNS servers - convert fully qualified domain names to IP addresses (e.g. when you address a web site, like http://www.w3schools.com, the name is translated to a number by a Domain Name Server (DNS).)

Example DNS translation
http://people.scs.carleton.ca/~ldnel/2405winter2011
http://134.117.29.41/~ldnel/2405winter2011

Parts of URL

https://google.com/#q=express
http://www.bing.com/search?q=grunt&first=9
http://localhost:3000/about?test=1#history

protocol	hostname	port	path	querystring	fragment
http://	localhost	:3000	/about	?test=1	#history
http://	www.bing.com		/search	?q=grunt&first=9	
https://	google.com		/		#q=express

Protocol
The protocol determines how the request will be transmitted (most common is http or https). Other common protocols ftp, telnet, mailto, etc.

Host
The host identifies the server. Servers on your computer (localhost) or a local network may simply be one word, or it may be a numeric IP address. On the Internet, the host will end in a top-level domain (TLD) like .com or .net. Additionally, there may be subdomains, which prefix the hostname. www is a very common subdomain, though it can be anything. Subdomains are optional.

Port
Each server has a collection of numbered ports. Some port numbers are “special,” like 80 and 443. If you omit the port, port 80 is assumed for HTTP and 443 for HTTPS. In general, if you aren’t using port 80 or 443, you should use a port number greater than 1023 because ports (0-1024) are reserved for standard services. It’s very common to use easy-to-remember port numbers like 3000, 8080, and 8088.

Path
The path is generally the first part of the URL that your app cares about. The path should be used to uniquely identify pages or other resources in your app. For example: http://carleton.ca/about about is a path. Even “/” is a path.

Query String
Query String is an optional collection of name/value pairs/ The query string starts with a question mark(?), and name/value pairs are separated by ampersands (&). Both names and values should be URL encoded.

JavaScript provides a built-in function to do that: encodeURIComponent. For example, spaces will be replaced with plus signs (+). Other special characters will be replaced with numeric character references.

https://google.com/#q=express
http://www.bing.com/search?q=grunt&first=9
http://localhost:3000/about?test=1#history

protocol	hostname	port	path	querystring	fragment
http://	localhost	:3000	/about	?test=1	#history
http://	www.bing.com		/search	?q=grunt&first=9	
https://	google.com		/		#q=express

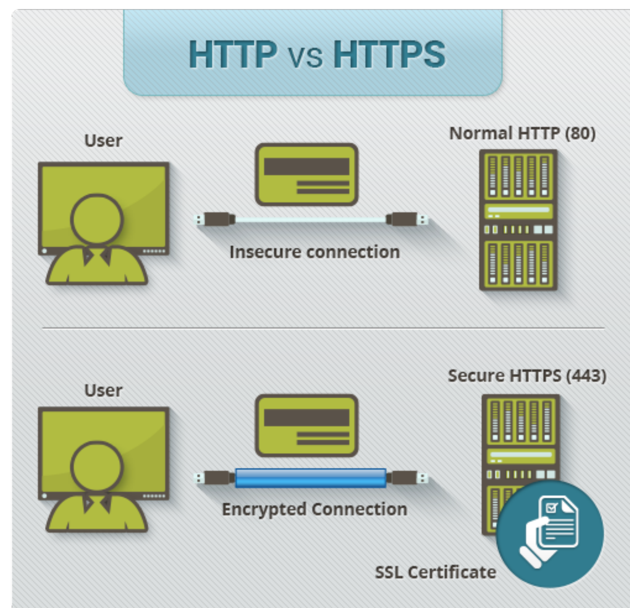
Fragment
The fragment (or hash) is not passed to the server at all: it is strictly for use by the browser. It is becoming increasingly common for single-page applications or AJAX-heavy applications to use the fragment to control the application. Originally, the fragment’s sole **purpose was to cause the browser to display a specific part of the document, marked by an anchor tag** (<a id=”history”>).

WEB SERVERS

- Provide responses to browser requests, either to deliver existing documents or dynamically build documents or data.
- Browser-server connection is now maintained through more than one request-response cycle
- Browser and Server can now establish a "data pipe" and transfer data asynchronously
- All communications between browsers and servers use Hypertext Transfer Protocol (HTTP)
- Web servers Monitor a communications port on the host, accepting HTTP messages when they appear

HTTP VS HTTPS

Hyper Text Transfer Protocol Secure (HTTPS) is the secure version of HTTP, the protocol over which data is sent between your browser and the website that you are connected to. The 'S' at the end of HTTPS stands for 'Secure'. It means all communications between your browser and the website are encrypted. HTTPS is often used to protect highly confidential online transactions like online banking and online shopping order forms. Web browsers such as Internet Explorer, Firefox and Chrome also display a padlock icon in the address bar to visually indicate that a HTTPS connection is in effect.



HTTPS based on combination of symmetric key and asymmetric (public) key cryptography. Encoding (or garbling) messages so that they cannot be read by unintended parties. Typical to have an encoding algorithm and decoding algorithm that uses a secret key. The **encoding and decoding algorithms are typically known publicly**. **It is the key that is kept secret.**

Symmetric Key Cryptography

Symmetric Key: sender and receiver use the same key.

- Typically, fast computations
- The key must transfer securely.
- Decoding algorithm typically known publicly so math must be hard to do without the key.
- Sufficiently hard to crack by brute force.

Example: Replace letter in message that is at position P in alphabet with one a position (P+k) mod 26. To decrypt replace letter in encrypted message at position C by letter at (C– k) mod 26.

```
Ciphertext: QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD
Plaintext:  THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG
```

Asymmetric Key Cryptography

Encryption and Decryption use a different key.

Public Key Cryptography

- The encryption key is public (everyone can see it)
- The encryption algorithm is public (everyone knows how the encrypted message is computed).
- Decryption uses a private key
- Decryption key should only be known by authorized parties.
- The decryption algorithm is public (everyone knows how the decryption algorithm works.)
- **Current methods make encryption and decryption computationally expensive.**
- **Comes from the fact they need to be hard to crack by brute force.**

If two parties already share a secret key, they could easily distribute new keys to each other by encrypting them with prior keys.

Public key cryptography is based on two facts and a conjecture from number theory. Most important part is conjecture.

To enable computers to encrypt data for a site, the site simply needs to publish it encryption key, for instance in a directory. Every computer can use that encryption key to protect data sent to the site. But only the site has the corresponding decryption key, so only it can decrypt the data.

Important: Self-signed certificate not trusted by browser

HTTPS Overall

- There are 3 different things in HTTPS: public key, private key and certificates.
- If you do not have private key, you cannot encrypt the messages.

- If you have self sign key the browser say tour connection is not private. Basically it says your key not stamp from lawyer.

The Secure Socket Layer protocol was created by Netscape to ensure secure transactions between web servers and browsers. The protocol uses a third party, a Certificate Authority (CA), to identify one end or both end of the transactions. This is in short how it works.

1. A browser requests a secure page (usually https://).
2. The web server sends its public key with its certificate.
3. The browser checks that the certificate was issued by a trusted party (usually a trusted root CA), that the certificate is still valid and that the certificate is related to the site contacted.
4. The browser then uses the public key, to encrypt a random symmetric encryption key and sends it to the server with the encrypted URL required as well as other encrypted http data.
5. The web server decrypts the symmetric encryption key using its private key and uses the symmetric key to decrypt the URL and http data.
6. The web server sends back the requested html document and http data encrypted with the symmetric key.
7. The browser decrypts the http data and html document using the symmetric key and displays the information.

SIMPLE HTTPS SERVER WITH NODE JS

```
var https = require("https");
var fs = require("fs");

//Private SSL key and signed certificate
var options = {
  key: fs.readFileSync('server.key'),
  cert: fs.readFileSync('server.crt')
};

var counter = 1;

https.createServer(options, function(req, res){
  console.log('REQUEST: ', req.method, " ", req.url);
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Req[' + counter++ + '] URL: ' + req.url + '\nHello Secure World\n');
}).listen(3000);

console.log('HTTPS Server listening at port: 3000  CNTL-C to quit');
```

BASIC STATIC SERVER

Node server does not try to analyze the URL to route the requests, it simply servers whatever files that happen to be in the ROOT_DIR directory. It does however replace a path of '/' with '/index.html'. It is because a static server, it does not analyze the requested URL. It simply serves such a file if it exists in the intended directory.

Static server uses the file extension of the requested resource to decide on the appropriate MIME type to return to the client.

If your server does not use MINE type (if it is not a static server), you can only serve simple html docs with imbedded images and imbedded CSS styles but fails when CSS styles are in separate files.

```
var http = require('http'); //need to http
var fs = require('fs'); //need to read static files
var url = require('url');  //to parse url strings


var ROOT_DIR = 'html'; //dir to serve static files from

var MIME_TYPES = {
    'css': 'text/css',
    'gif': 'image/gif',
    'htm': 'text/html',
```

```
    'html': 'text/html',
    'ico': 'image/x-icon',
    'jpeg': 'image/jpeg',
    'jpg': 'image/jpeg',
    'js': 'text/javascript', //should really be application/javascript
    'json': 'application/json',
    'png': 'image/png',
    'txt': 'text/text'
};

var get_mime = function(filename) {
    var ext, type;
    for (ext in MIME_TYPES) {
        type = MIME_TYPES[ext];
        if (filename.indexOf(ext, filename.length - ext.length) !== -1) {
            return type;
        }
    }
    return MIME_TYPES['txt'];
};

http.createServer(function (request,response){
     var urlObj = url.parse(request.url, true, false);
     console.log('\n============================');
     console.log("PATHNAME: " + urlObj.pathname);
     console.log("REQUEST: " + ROOT_DIR + urlObj.pathname);
     console.log("METHOD: " + request.method);

     var filePath = ROOT_DIR + urlObj.pathname;
     if(urlObj.pathname === '/') filePath = ROOT_DIR + '/index.html';

     fs.readFile(filePath, function(err,data){
       if(err){
          //report error to console
          console.log('ERROR: ' + JSON.stringify(err));
          //respond with not found 404 to client
          response.writeHead(404);
          response.end(JSON.stringify(err));
          return;
         }
         response.writeHead(200, {'Content-Type': get_mime(filePath)});
         response.end(data);
       });

 }).listen(3000);

console.log('Server Running at http://127.0.0.1:3000  CNTL-C to quit');
```

WHAT IS MIME? (Get mime function in the above code)

Multipurpose Internet Mail Extensions (MIME) originally developed for email. MIME uses to specify to the browser the form of the contents in a file returned by the server (attached by the server to the beginning of the document)

Type specifications: type/subtype. E.g. text/plain, text/html, image/gif, image/jpeg

Server infers type from the requested file name's suffix (.html implies text/html) and protocol. Browser gets the type explicitly from the server's response message.

Experimental types
Subtype begins with x (e.g., video/x-msvideo)
Experimental types require the server to send a helper application or plug-in so the browser can deal with the file

THICK vs. THIN CLIENT

A **thin client** is software that is primarily designed to communicate with a server. Its features are produced by servers such as a cloud platform.

A **thick client** is software that implements its own features. It may connect to servers but it remains mostly functional when disconnected. In thick client mostly works done by client but in thin client mostly works done by server (rendering page, dealing with data, sending html page, etc.). **Basically, thin client server does all the works (e.g. rendering web page) and in the thick client, client expect data from server and client is doing all work.**

Thin Client vs Thick Client		
	Thin Client	**Thick Client**
Definition	Application that relies on a server.	Application that runs at least some features directly on your device.
Offline	Functions mostly don't work	Functions mostly work
Local Resources	Generally consumes few local resources such as disk, computing power and memory.	Generally consumes more local resources
Network Latency	Functionality may depend on a fast network connection.	Functionality may work without a connection or with a slow connection.
Data	Data is typically stored on servers.	Data may be stored locally.

HTTP METHODS: GET vs. POST

Two commonly used methods for a request-response between a client and server are: GET and POST.

The GET Method

The query string (name/value pairs) is sent in the URL of a GET request

```
/test/demo_form.php?name1=value1&name2=value2
```

- GET requests can be cached
- GET requests remain in the browser history
- GET requests can be bookmarked
- GET requests should never be used when dealing with sensitive data
- GET requests have length restrictions
- GET requests should be used only to retrieve data

The POST Method

The query string (name/value pairs) is sent in the HTTP message body of a POST request

```
POST /test/demo_form.php HTTP/1.1
Host: w3schools.com
name1=value1&name2=value2
```

- POST requests are never cached
- POST requests do not remain in the browser history
- POST requests cannot be bookmarked
- POST requests have no restrictions on data length

Other HTTP Request Methods

HEAD Same as GET but returns only HTTP headers and no document body

PUT Uploads a representation of the specified URI

DELETE Deletes the specified resource

OPTIONS Returns the HTTP methods that the server supports

CONNECT Converts the request connection to a transparent TCP/IP tunnel

Compare GET vs. POST		
BACK button/Reload	Harmless	Data will be re-submitted (The browser should alert the user that the data are about to be re-submitted)
Bookmarked	Can be bookmarked	Cannot be bookmarked
Cached	Can be cached	Not cached
Encoding type	application/x-www-form-urlencoded	application/x-www-form-urlencoded or multipart/form-data. Use multipart encoding for binary data
Browser history	Parameters remain in browser history (Parameters in the URL)	Parameters are not saved in browser history (Parameters in the body)
Restriction on data length	Yes, when sending data, the GET method adds the data to the URL; and the length of a URL is limited (Max 2048 characters)	No restrictions (You can send as many as)
Restrictions on data type	Only ASCII characters allowed	No restrictions. Binary data is also allowed.
Security	GET is less secure compared to POST because data sent is part of URL Never use GET when sending passwords or other sensitive information! (Login form should be post request)	POST is a little safer than GET because the parameters are not stored in browser history or in web server logs.
Visibility	Data is visible to everyone in the URL	Data is not displayed in the URL

HTTP REQUESTS

Form:

HTTP method domain part of URL HTTP ver.
Header fields
blank line
Message body

An example of the first line of a request:

GET /cs.uccp.edu/degrees.html HTTP/1.1

HTTP RESPONSES

- Header and Body separated by empty line
- Header must state content-type
- Example:
  - HTTP/1.0 200 OK
    Date: Fri, 27 Aug 2004 10:05:30 GMT
    Server: Apache/1.2.13 (Linux)
    Last-Modified: Thu, 26 Aug 2004 20:14:26 GMT
    Content-Length: 2523
    Content-Type: text/html

    <html> <head> <title>COMP 2405 Internet Applications</title> ...

Response Code

HTTP/1.0 **200** OK

Response may come with a code number

- 1xx = information
- 2xx = success
- 3xx = redirection
- 4xx = client error (e.g. common 404 “Not Found” error)
- 5xx = server error

An example of a complete response header:

HTTP/1.1 200 OK
Date: Sat, 25 July 2009 20:15:11 GMT
Server: Apache /2.2.3 (CentOS)
Last-modified: Tues, 18 May 2004 16:38:38 GMT
Etag: "1b48098-16a-3dab592dc9f80"
Accept-ranges: bytes

Content-length: 364
Connection: close
Content-type: text/html, charset=UTF-8

Both request headers and response headers must be followed by a blank line.

SYNCHRONOUS VS ASYNCHRONOUS FUNCTIONS

It is very typical in JavaScript environments (Browsers, Node.js) to provide asynchronous functions. The Node.js execution environment provides many utility functions that are asynchronous functions taking a single callback parameter.

```
// Example 1 - Synchronous (blocks)
var result = database.query("SELECT * FROM hugetable");
console.log("Query finished");
console.log("Next line");
```

Output

```
Query finished
Next line
```

```
// Example 2 - Asynchronous (doesn't block)
database.query("SELECT * FROM hugetable", function(result) {
    console.log("Query finished");
});
console.log("Next line");
```

Output

```
Next line
Query finished
```

Synchronous Function: It waits for each operation to complete, after that only it executes the next operation. The console.log() command will not be executed until & unless the query has finished executing to get all the result from Database.

Asynchronous Function: It never waits for each operation to complete, rather it executes all operations in the first GO only. The result of each operation will be handled once the result is available. The console.log() command will be executed soon after the

Database.Query() method. While the Database query runs in the background and loads the result once it is finished retrieving the data.

MEANING OF "CALLBACK" FUNCTIONS

A callback function is a function that is passed to another function (let's call this other function "otherFunction") as a parameter, and the callback function is called (or executed) inside the otherFunction. A callback function is essentially a pattern (an established solution to a common problem), and therefore, the use of a callback function is also known as a callback pattern.

```
function mySandwich(param1, param2, callback) {
    console.log('Started eating my sandwich.\n\nIt has: ' + param1 + ', ' + param2);
    callback();
}

mySandwich('ham', 'cheese', function() {
    console.log('Finished eating my sandwich.');
});
```

Output

```
Started eating my sandwich.

It has: ham, cheese
Finished eating my sandwich.
```

We can pass functions around like variables and return them in functions and use them in other functions. When we pass a callback function as an argument to another function, we are only passing the function definition. We are not executing the function in the parameter. In other words, we aren't passing the function with the trailing pair of executing parenthesis () like we do when we are executing a function.

When we pass a callback function as an argument to another function, the callback is executed at some point inside the containing function's body just as if the callback were defined in the containing function. This means the **callback is a closure.**

PART II
JavaScript Programming

== vs === OPERATOR,

VAR vs LET vs CONST,

ARRAYS AND STRINGS, JAVASCRIPT OBJECTS

MEANING OF “THIS” KEYWORD,

MEANING OF PROTOTYPES,

JAVASCRIPT MODULES AND THE EXPORTS KEYWORD,

MEANING OF “CALLBACK” FUNCTIONS,

MEANING OF CLOSURES

== vs === OPERATOR

The identity (===) operator behaves identically to the equality (==) operator except no type conversion is done, and the types must be the same to be considered equal.

For example, “5” == 5 is true but “5” === 5 is not true. “==” operator does not care the number is string or not.

The == operator will compare for equality after doing any necessary type conversions. The === operator will not do the conversion, so if two values are not the same type === will simply return false

Overall;
Using the == operator (Equality)
true == 1; //true, because 'true' is converted to 1 and then compared
"2" == 2; //true, because "2" is converted to 2 and then compared

Using the === operator (Identity)
true === 1; //false
"2" === 2; //false
2 === 2; //true

== equal to
!= not equal
=== equal value and equal type
!== not equal value or not equal type

This is because the equality operator == does type coercion, meaning that the interpreter implicitly tries to convert the values before comparing. On the other hand, the identity operator === does not do type coercion, and thus does not convert the values when comparing.

== allows the compiler to type-cast its arguments but === does not allow type casting.

VAR vs LET vs CONST,

Variables declared implicitly (without var keyword), even locally in functions, have GLOBAL scope (become global variables). This creates bugs that are hard to track down.

In JavaScript undefined and NaN are not constants. They are global variables and you can change their values

- Variable declaration is "hoisted" to top of function scope (not block scope).
    - In other words; a variable can be used before it has been declared.
- Initialization of variable is not hoisted.
    - https://www.w3schools.com/js/js_hoisting.asp
- Can be source of many bugs
- Variables declared with let are not hoisted and have block, not function scope
- Variables declared with var have function scope but those declared with let have block scope.
- JavaScript 6 has let and const in addition to var

In JavaScript, "**const**" means that the identifier can't be reassigned. "const" is a signal that the identifier won't be reassigned.

"**let**", is a signal that the variable may be reassigned, such as a counter in a loop, or a value swap in an algorithm. It also signals that the variable will be used only in the block it's defined in, which is not always the entire containing function. "let" allows you to declare variables that are limited in scope to the block, statement, or expression on which it is used.

The **var** statement declares a variable. Use the var keyword only for declaration or initialization, once for the life of any variable name in a document. You should not re-declare same variable twice.

Variables are containers for storing information and creating a variable in JavaScript is called "declaring" a variable.

Variable names can contain letters, digits, underscores, and dollar signs.

- Variable names must begin with a letter
- Variable names can also begin with $ and _
- Dollar sign is "special" in JavaScript but unfortunately doesn't do magic.
- Subsequent characters may be letters, digits, underscores, or dollar signs.
    - 123test is an invalid variable name but _123test, $123test is a valid one.
- Variable names are case sensitive (y and Y are different variables)
- Reserved words (like JavaScript keywords) cannot be used as variable names

- Value is optional
- Note: A variable declared without a value will have the value undefined

You can declare many variables in one statement. Start the statement with var and separate the variables by comma:

```
var lastName = "Doe",
age = 30,
job = "carpenter";
```

```
function varTest() {
    var x = 1;
    if (true) {
        var x = 2;  // same variable!
        console.log(x);  // 2
    }
    console.log(x);  // 2
}

function letTest() {
    let x = 1;
    if (true) {
        let x = 2;  // different variable
        console.log(x);  // 2
    }
    console.log(x);  // 1
}


// Let and var example

var i // even with let i result will be same
i = 34
for(let i =0; i<4; i++){
 console.log(i) // 1,2,3
}
console.log(i) // 34

for(let i =0; i<4; i++){
    console.log(i) // 1,2,3
}
console.log(i) // error let i defined only inside the for loop

for(i =0; i<4; i++){
    console.log(i) // 1,2,3
```

```
}
console.log(i) // 4  - it is var which is global
```

JavaScript Variable Scope

The scope of a variable is the region of your program in which it is defined. JavaScript variables have only two scopes.

- **Global Variables –** A global variable has global scope which means it can be defined anywhere in your JavaScript code.

- **Local Variables –** A local variable will be visible only within a function where it is defined. Function parameters are always local to that function.

```
var myVar = "global"; // Declare a global variable
function checkscope( ) {
  var myVar = "local";  // Declare a local variable
  console.log("myVar inside function: ", myVar);
}

checkscope();
console.log("myVar outside: " + myVar)
```

Output

```
myVar inside function:  local
myVar outside: global
```

Legal and Illegal

var method; // ok
var class; // illegal
function for(){return 42;}; // illegal
object = {box: value}; // ok
object = {case: value}; //ok
object = {'case': value}; // ok
object.box = value; // ok
object.case = value; //ok
object['case'] = value; // ok
object[case] = value; // illegal

ARRAYS AND STRINGS

What is an array

An array is a special variable, which can hold more than one value at a time. The JavaScript Array object is a global object that is used in the construction of arrays; which are high-level, list-like objects. Using an array literal is the easiest way to create a JavaScript Array.

```
var array_name = [item1, item2, ...];
```

Example:

```
// Creating an array
var fruits = ['Apple', 'Banana'];
console.log(fruits.length); // 2


// Access (index into) an Array item
var first = fruits[0]; // Apple
var last = fruits[fruits.length - 1]; // Banana
var lastPlusOne = fruits[fruits.length]; // Undefined


// Loop over an Array
fruits.forEach(function(item, index, array) {
    console.log(item, index);
});
// Apple 0
// Banana 1


// Another loop
for(var i = 0; i<fruits.length; i++){
    console.log(fruits[i]);
}
// Apple
// Banada


fruits.name = "Watermelon"
console.log("\n\n");
for(var i = 0; i<fruits.length; i++){
    console.log(fruits[i]);
}
// Apple
// Banana
```

```
// Do not use for—in loop because If
// properties are added to the string or array the
// loop will loop over those as well.
for(k in fruits) console.log(fruits[k]);
// Apple
// Banana
// Watermelon


// length of an array
console.log(fruits.length); // 2


// Add to the end of an Array
var newLength = fruits.push('Orange');
// ["Apple", "Banana", "Orange"]


// Remove from the end of an Array
var last = fruits.pop(); // remove Orange (from the end)
// ["Apple", "Banana"];


// Remove from the front of an Array
var first = fruits.shift(); // remove Apple from the front
// ["Banana"];

// Add to the front of an Array
var newLength = fruits.unshift('Strawberry') // add to the front
// ["Strawberry", "Banana"];


//Find the index of an item in the Array
// ["Banana", "Mango"]
var pos = fruits.indexOf('Mango');
// 1


// Add another item
fruits.push('Mango');
// ["Strawberry", "Banana", "Mango"]


// Remove an item by index position
var removedItem = fruits.splice(pos, 1); // this is how to remove an item
// ["Strawberry", "Mango"]
```

```
// Copy an Array
var shallowCopy = fruits.slice(); // this is how to make a copy
// ["Strawberry", "Mango"]
```

Important about array

Adding elements with high indexes can create undefined "holes" in an array:

```
var names = [];
names[1] = "John";
names[8] = "Matt"
names["Cuneyt"] = "Celebican";

console.log(names.length); // 9
console.log(names[8].length); // 4

console.log(names); // (9) [empty, "John", empty × 6, "Matt", Cuneyt: "Celebican"]

console.log(names.1); // a syntax error
```

You can also create an array with “new” keyword but avoid to use it. Use [ ] instead.

```
var names = new Array("Saab", "Volvo", "BMW");

console.log(names.length); // 3
console.log(names[1].length); // 5

console.log(names); // (3) ["Saab", "Volvo", "BMW"]
```

Because it complicates the code.

```
var points = new Array(40, 100);
// Creates an array with two elements (40 and 100)

var points = new Array(40);
// Creates an array with 40 undefined elements !!!!!
```

Arrays are Objects

Arrays are a special type of objects. The typeof operator in JavaScript returns "object" for arrays.

```
var cars = {car1:"Saab", car2:"Volvo", car3:"BMW"};
console.log(cars.car1); // Saab
console.log(cars); // {car1: "Saab", car2: "Volvo", car3: "BMW"}
console.log(cars.length); // undefined
console.log(cars[0]); // undefined
```

If you use named indexes, JavaScript will redefine the array to a standard object. After that, some array methods and properties will produce incorrect results.

typeof operator does not distinguish between arrays and objects

Array Elements Can Be Objects

JavaScript variables can be objects. Arrays are special kinds of objects. Because of this, you can have variables of different types in the same Array. You can have objects in an Array. You can have functions in an Array. You can have arrays in an Array:

```
myArray[0] = Date.now;
myArray[1] = myFunction;
myArray[2] = myCars;
```

The length Property

The length property of an array returns the length of an array (the number of array elements). Length start to count from 1.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.length; // 4
```

Array Sort Method

The sort() method sorts arrays

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
```

```
console.log(fruits.length); // 4
console.log(fruits); // (4) ["Banana", "Orange", "Apple", "Mango"]
console.log(fruits.sort()); // (4) ["Apple", "Banana", "Mango", "Orange"]
```

STRINGS

A JavaScript string stores a series of characters like "John Doe". A string can be any text inside double or single quotes:

```
var carname = "Volvo XC60";
var carname = 'Volvo XC60';
```

String indexes are zero-based: The first character is in position 0, the second in 1, and so on. The escape character for string is “\”

String as a object

You can use “new” keyword to create JavaScript string object

```
var john = new String("John");
console.log(john) // String {"John"}
john.lastname = "Deo"
console.log(john); // String {"John", lastname: "Deo"}
console.log(john.lastname); // Deo
```

Do not get confused

```
var x = "John";
var y = new String("John");
var z = new String("John");
// (x == y) is true because x and y have equal values
// (x === y) is false because x and y have different types (string and object)
// (y == z) is false because x and y are different objects
// (x === y) is false because x and y are different objects
```

Note the difference between (x==y) and (x===y). Comparing two JavaScript objects will always return false

SOME STRING METHODS

String split() Method

Split a string into an array of substrings:

```
var days="Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday";
var daysArray = days.split(",");
console.log(daysArray);
//(7) ["Monday", " Tuesday", " Wednesday", " Thursday", " Friday", " Saturday", " Sunday"]
```

String slice() Method

Extract parts of a string:

```
var str = "Hello world!";
var res = str.slice(1, 5); // ello
```

String charAt() Method

Return the character in the given position:

```
var str = "HELLO WORLD";
var res = str.charAt(0); // H
```

String startsWith() Method

Returns true if string starts with given parameter.

```
var str = "Hello world, welcome to the universe.";
var n = str.startsWith("Hello"); //True
```

String substring() Method

Extract characters from a string:

```
var str = "Hello world!";
var res = str.substring(1, 4); //ell
```

JavaScript String substr() Method

Extract parts of a string:

```
var str = "Hello world!";
var res = str.substr(1, 4); //ell
```

JAVASCRIPT OBJECTS

In real life, a car is an object. A car has properties like weight and color, and methods like start and stop. All cars have the same properties, but the property values differ from car to car. All cars have the same methods, but the methods are performed at different times.

Object	Properties	Methods
	car.name = Fiat	car.start()
	car.model = 500	car.drive()
	car.weight = 850kg	car.brake()
	car.color = white	car.stop()

You have already learned that JavaScript variables are containers for data values. This code assigns a simple value (Fiat) to a variable named car:

```
var car = "Fiat";
```

Objects are variables too. But objects can contain many values (like an array). This code assigns many values (Fiat, 500, white) to a variable named car:

```
var car = {type:"Fiat", model:"500", color:"white"};
```

The values are written as name:value pairs (name and value separated by a colon). The name:values pairs (in JavaScript objects) are called properties.

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

Property	Property Value
firstName	John
lastName	Doe
age	50
eyeColor	blue

Accessing Object Properties

You can access object properties in two ways:

objectName.propertyName

or

objectName["propertyName"]

Example: (Both will produce same output)

```
person.lastName;
person["lastName"];
```

Legal and Illegal Accessing to Object Properties

```
x.name; //'Lou'
x[name]; //ERROR
x['name']; //'Lou'
x["name"]; //'Lou'
```

Object-Based: Javascript

- There are only objects, no classes (ES6 adds classes).
- Objects are created as individuals and need not be related to any type.
- "Inheritance" is object-based. An object can inherit from another object. (but in JS inheritance is just objects pointing to other objects –lots of confusion here.)
- New kinds of objects can be created without defining a type.

IMPORTANT NOTES

- JavaScript has two special kinds of objects: function objects and arrays. (functions and arrays are objects in JavaScript and have some additional capabilities:).
- In JavaScript anything that is not: number, string, Boolean, NaN or undefined is an object and always referred to by reference.
- Function objects are callable, and array objects are indexable by integers, otherwise they are still just regular JavaScript objects.
- Objects retain a connection to their ancestor prototypes which are themselves objects.
- Objects do not derive from classes (types) but rather from reference links to other objects. **Consequences:**
    - To add and remove properties to existing object you don't need to change, and recompile, a class definition
    - You can add properties to existing objects and also remove them at any time.
    - You can add "methods" to existing objects or remove them. (JavaScript does not really have methods –adding a function to an object does not make it a method)

MEANING OF “THIS” KEYWORD

In the global execution context (outside of any function), this refers to the global object whether in strict mode or not.

Where a function uses the this keyword in its body, its value can be bound to a particular object in the call using the call or apply methods which all functions inherit from Function.prototype.

What does “this” refer in this context?

```
var obj = {size: "Big"}
var func = function(x){this.colour = x};
func.sound = 'loud'
(func.__proto__).taste = 'sweet';
func.prototype.smell = 'stinky';
var y = func('yellow');
```

“this” refers to the global object

You can also implement like this

```
var func = function(x){this.colour = x};
var y = {func, name:"John"};
y.func('yellow');
console.log(y);
```

MEANING OF PROTOTYPE

JavaScript has a hidden property within objects called [[Prototype]] to refer to the inheritance prototype of the object.

Two ways of accessing the prototype:
obj.__proto__;
and
Object.getPrototypeOf(obj);

There is also an obj.prototype property but that only applies if obj is a function, otherwise obj.prototype is undefined (does not exist in non-function objects).

Very confusing:
obj.prototype is not the inheritance prototype of obj.
.prototype turns out to be a very poorly chosen name for what .prototype actually represents.

Objects "inherit" the properties of their prototypes in that they are linked, by reference, to their prototypes. Logging an object to the console does not reveal its inherited properties –important to remember since console.log() will be used often for debugging and inspection.

All JavaScript objects inherit the properties and methods from their prototype. Objects created with new Date() inherit the Date.prototype. The Object.prototype is on the top of the prototype chain.

Creating a Prototype

The standard way to create an object prototype is to use an object constructor function:

```
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
}
```

With a constructor function, you can use the **new** keyword to create new objects from the same prototype:

```
var myFather = new Person("John", "Doe", 50, "blue");
var myMother = new Person("Sally", "Rally", 48, "green")
```

You can also create with curly bracket:

```
var x = {name: 'Lou'};
console.log(x); // {name: "Lou"}
x.name = 'Louis'; //re-assign
x.email = 'ldnel@scs.carleton.ca'; //create
x['office'] = '5370 Herzberg'; //create
console.log(x);
// {name: "Louis", email: "ldnel@scs.carleton.ca", office: "5370 Herzberg"}
```

Adding a new property to an existing object is easy:

```
myFather.nationality = "English";
```

Adding a new method to an existing object is also easy:

```
myFather.name = function () {
    return this.firstName + " " + this.lastName;
};
```

Adding Properties to a Prototype

You cannot add a new property to a prototype the same way as you add a new property to an existing object, because the prototype is not an existing object.

```
Person.nationality = "English";
```

To add a new property to a prototype, you must add it to the constructor function:

```
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
    this.nationality = "English";
}
```

Prototype properties can have prototype values (default values).

Adding Methods to a Prototype

Your constructor function can also define methods:

```
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
    this.name = function() {return this.firstName + " " + this.lastName;};
}
```

Adding methods to an object is done inside the constructor function:

```
function Person(firstName, lastName, age, eyeColor) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
    this.eyeColor = eyeColor;
    this.changeName = function (name) {
        this.lastName = name;
    };
}
```

The changeName() function assigns the value of name to the person's lastName property.

```
myMother.changeName("Doe");
```

What happens below if the new is forgotten when trying to create a plane?

```
function plane(newMake, newModel, newYear){
    this.make = newMake;
    this.model = newModel;
```

```
    this.year = newYear;
}
myPlane = new plane("Cessna", "Centurian", "1970");
```

Answer: the “this” variable in the constructor binds with the global Object, from which everyone inherits, and the constructor then proceeds to clobber this object with new property values --this is very bad!

Using the prototype Property

The JavaScript prototype property allows you to add new properties to an existing prototype:

```
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
}
Person.prototype.nationality = "English";
```

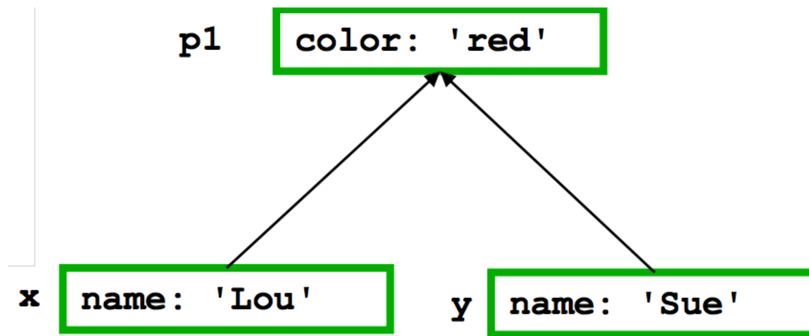
For Loop with Prototype

You can loop through the objects properties with a for-in loop and this will pick up the properties of the prototype. This will be very useful to ensure you have examined the whole object: it’s immediate properties and those inherited through prototypes.

```
var p1 = {color: 'red'};
var x = {name: 'Lou'}
x.__proto__ = p1;
for(k in x) console.log(k);
// name
// color
for(k in x) console.log(k + ": " + x[k]);
// name: Lou
// color: red
```

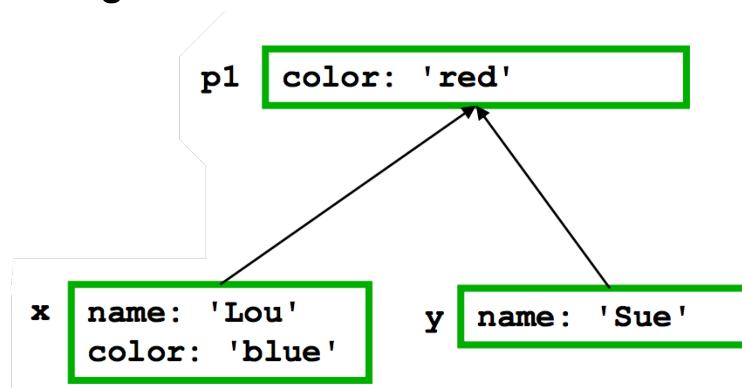
Prototype Objects Are Shared by Reference

Prototypes are often shared by reference. Helpful to simulate “classical” inheritance.




```
var x = {name: 'Louis'};
x.__proto__ = p1;
var y = {name: 'Sue'};
y.__proto__ = p1;
x; //{ name: 'Lou' }
y; //{ name: 'Sue' }
for(k in x) console.log(k + ": " + x[k]);
//name: Lou
// color: red
for(k in y) console.log(k + ": " + y[k]);
// name: Sue
// color: red
```

Property Shadowing




```
var x = {name: 'Louis'};
x.__proto__ = p1;
var y = {name: 'Sue'};
y.__proto__ = p1;
x; //{ name: 'Lou' }
y; //{ name: 'Sue' }
for(k in x) console.log(k + ": " + x[k]);
//name: Lou
// color: red
for(k in y) console.log(k + ": " + y[k]);
// name: Sue
```

```
// color: red
x.color = 'blue';
for(k in x) console.log(k+": "+x[k]);
//name: Lou
//color: blue
for(k in y) console.log(k+": "+y[k]);
//name: Sue
//color: red

x.hasOwnProperty('color'); //true
y.hasOwnProperty('color'); //false
```

Note this is the default behavior but can be overridden with custom getter/setter property functions

MEANING OF "CALLBACK" FUNCTIONS

A callback function is a function that is passed to another function (let's call this other function "otherFunction") as a parameter, and the callback function is called (or executed) inside the otherFunction. A callback function is essentially a pattern (an established solution to a common problem), and therefore, the use of a callback function is also known as a callback pattern.

```
function mySandwich(param1, param2, callback) {
    console.log('Started eating my sandwich.\n\nIt has: ' + param1 + ', ' + param2);
    callback();
}

mySandwich('ham', 'cheese', function() {
    console.log('Finished eating my sandwich.');
});
```

Output

```
Started eating my sandwich.

It has: ham, cheese
Finished eating my sandwich.
```

We can pass functions around like variables and return them in functions and use them in other functions. When we pass a callback function as an argument to another function, we are only passing the function definition. We are not executing the function in the parameter. In other words, we aren't passing the function with the trailing pair of executing parenthesis () like we do when we are executing a function.

When we pass a callback function as an argument to another function, the callback is executed at some point inside the containing function's body just as if the callback were defined in the containing function. This means the **callback is a closure.**

MEANING OF "CLOSURES" FUNCTIONS

JavaScript variables can belong to the local or global scope. Global variables can be made local (private) with closures.
https://www.w3schools.com/js/js_function_closures.asp

```
var counter = 100; //line 1
function make(){
 var local = 10;
 return function(x){return x + local + counter;} //line 2
}
var action = make(); //line 3
console.log(action(8)); //line 4
```

It will print 118

SCOPE

- In JavaScript, {blocks} do not have scope
- Only functions have scope
- Variables defined in a function are not visible outside the function
- Variables are hoisted to top of functions
- Variables are hoisted out of blocks to top of enclosing function

PART III
Node and Express

CODING A BASIC STATIC SERVER WITH NODE

USING CORRECT MIME TYPES,

ASYNCHRONOUS FILE READING

JAVASCRIPT MODULES AND THE EXPORTS KEYWORD

MEANING AND USE OF PACKAGE.JSON AND NPM

MEANING OF API (AND RESTFUL API),

TEMPLATE RENDERING

USING JSON DATA (JSON.stringify() and JSON.parse())

EXPRESS ROUTES AND MIDDLEWARE

CODING A BASIC STATIC SERVER WITH NODE

```
var http = require('http');
var ecStatic = require('ecstatic');  //provides static file server service
var server = http.createServer(ecStatic({root: __dirname + '/www'}));
server.listen(3000);
console.log('Server Running at http://127.0.0.1:3000/assignment2.html  CNTL-C to
quit');
```

ASYNCHRONOUS FILE READING

The "normal" way in Node.js is probably to read in the content of a file in a non-blocking, asynchronous way. That is, to tell Node to read in the file, and then to get a callback when the file-reading has been finished. That would allow us to hand several requests in parallel.

```
var fs = require('fs');
fs.readFile('DATA', 'utf8', function(err, contents) {
   console.log(contents);
});
console.log('after calling readFile');
```

First we load the fs class using the require command. Then we call the readFile method that gets 3 parameters: The name of the file ('DATA' in this case), the encoding of the file ('utf8' in the examples), and a function. This function is going to be called when the file-reading operation has finished. The function will get two parameters. The first is the information about any error conditions, the second is the actual content of the file.

Once this is called Node starts to read the file in the background, but it also keeps executing our program. That is, it will call the console.log('after calling readFile'); and will print that text to the console. Then, once the file has been read into memory, Node will run the function we provided to the readFile method and that will print the content of the file.

Read file in synchronously (blocking)

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

var contents = fs.readFileSync('DATA', 'utf8');
console.log(contents);
```

MODULES AND THE EXPORTS KEYWORD

Consider modules to be the same as JavaScript libraries. A set of functions you want to include in your application.

Use the **exports** keyword to make properties and methods available outside the module file.

Node.js provides three kinds of modules

- Core modules –ship with node.js
- File based modules –loaded from local file system
- Imported from npm registry

To include a module, use the require() function with the name of the module:

```
require('bar');
```

Look for core modules with the same name, for example, bar. If no core module matching this name is found, we look for an imported node_module called 'bar'.

Node.js has a set of built-in modules which you can use without any further installation. For example, fs, http, https, path, url, etc.

You can include a module using variable and you can use its features.

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

Now your application has access to the HTTP module, and is able to create a server:

```
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end('Hello World!');
}).listen(8080);
```

Create Your Own Modules (File-Based Modules)

You can create your own modules, and easily include them in your applications. File-based module is loaded from the specified path.
For example;

- require('./bar);
- require('../bar/bar');
- require('/full/path/to/a/node/module/file');

Scanning order for imported Node Modules

/home/ryo/project/node_modules/bar.js
/home/ryo/node_modules/bar.js
/home/node_modules/bar.js
/node_modules/bar.js

In other words, Node.js looks for 'node_modules/bar.js' in the current folder followed by every parent folder until it reaches the root of the file system tree for the current file or until a bar.js is found.

It is common to have a several files working toward a singular goal. Node.js has explicit support for this mechanism. If a path to the module resolves to a folder (instead of a file), Node.js will look for an index.js file in that folder and return that as the module file.

You can also save your file inside the node_modules file and you can easily call it with following:

```
require('foo');
```

In Node.js, each module can have its own node_modules folder and different versions of moduleZ can coexist. Modules do not need to be global in Node.js

```
projectroot
l--
        node_modules/foo/index.js
                l-- moduleA/ a.js
                l-- moduleB/ b.js
```

If a.js and b.js both require('foo') they will get the same cached module export result.

```
projectroot
l--
        node_modules/foo/index.js
                l-- moduleA/a.js
                l-- moduleB
                        l--
                        node_modules/foo/index.js
                        l-- b.js
```

If a.js and b.js both require('foo') this time they will get the different versions of module foo.

In Node.js, if a foo.js is not found as a result of a require('foo'), Node.js will look for a foo.json foo.json will be treated as a JSON string and converted to a javascript object and returned as the export of the module.

```
//config.json
{"foo": "this is the value for foo"}

//app.js
var config = require('./config');
console.log(config.foo); //this is the value for foo
```

CREATING OWN MODULE

The following example creates a module that returns a date and time object:

```
exports.myDateTime = function () {
    return Date();
};
```

Use the **exports** keyword to make properties and methods available outside the module file. Save the code above in a file called "myfirstmodule.js"

Include Your Own Module

Now you can include and use the module in any of your Node.js files.

```
var http = require('http');
var dt = require('./myfirstmodule');

http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write("The date and time are currently: " + dt.myDateTime());
    res.end();
}).listen(8080);
```

MEANING AND USE OF PACKAGE.JSON AND NPM

Package.json

A package.json file contains meta data about your app or module. Most importantly, it includes the list of dependencies to install from npm when running npm install.

npm init

npm init is a bash command for creatin a package.json dependencies file

npm install

You can install npm modules with following command in the terminal:

```
npm install underscore
```

it does not effect your package.json file. If you type following then you'll add the npm module with version number to your package.json file.

For npm install you do not need any package.json file. However npm install <<package>> -- save (or -save) command you need a package.json file.

When you use –save(or -save) you are saving under the dependencies in the package.json file. In addition, if you do not specify the version, npm always install latest version of the module.

npm ls

Listing npm dependencies

npm rm

removing a npm dependencies

```
npm rm underscore
```

If you also want to delete from package.json file

```
npm rm underscore --save
```

Installing all dependencies in your package.json

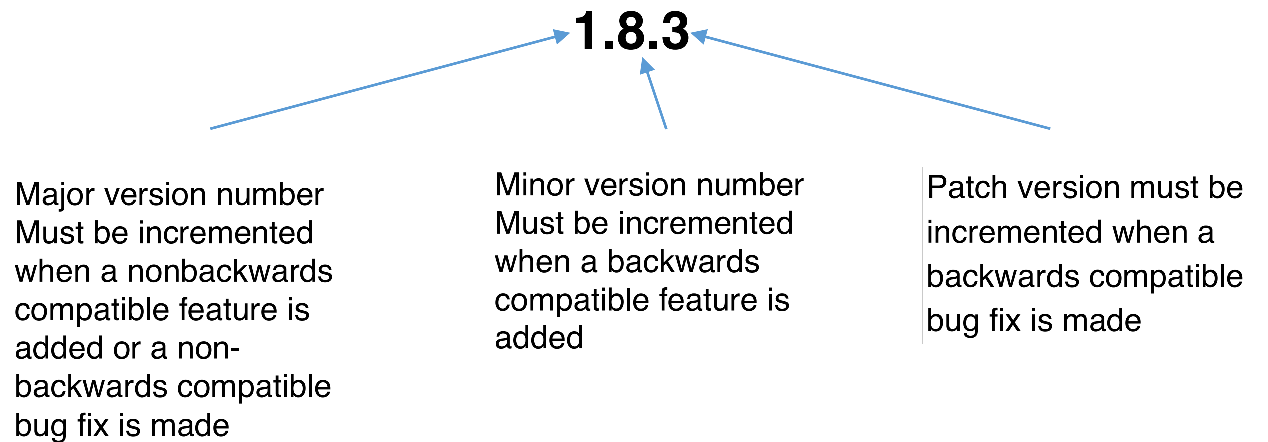
npm install

Global installation vs Local installation

If you want to use it as a command line tool, something like the grunt CLI, then you want to install it globally. On the other hand, if you want to depend on the package from your own module using something like Node's require, then you want to install locally. To download packages globally, you simply use the command

```
npm install -g <package>
```

NPM VERSION NUMBERS
Npm uses the following Semantic (meaningful) version numbering scheme



~1.8.3
~ means latest patch compatible version e.g. 1.8.5 But not 1.9.0 or 2.0.1

^1.8.3
^ means latest minor version compatible version e.g. 1.9.5 but not 2.0.1

Other versions
1.8.*
>=1.8.9
<2.0.3
* //latest version

Installing specific versions
> npm install underscore@1.0.3
> npm install underscore@"~1.0.0"
> npm install underscore@"^1.0.0"
> npm install underscore //latest version

You can also update the package.json

```
npm install underscore@"^1.0.0" –save
```

Check Node modules
http://people.scs.carleton.ca/~comp2406/notes/13%20Node%20modules%20and%20Data%20Structures%20rev1/

MEANING OF API (AND RESTFUL API)

API is the acronym for Application Programming Interface, which is a software intermediary that allows two applications to talk to each other. Each time you use an app like Facebook, send an instant message, or check the weather on your phone, you're using an API.

RESTFUL API

An architectural style called REST (Representational State Transfer) advocates that web applications should use HTTP as it was originally envisioned. Lookups should use GET requests. PUT, POST, and DELETE requests should be used for mutation, creation, and deletion respectively.

```
/*
Interacting with external services

Simple example of node.js app serving contents based
on an available internet service.
In this case api.openweathermap.org

***IMPORTANT NOTE***
As of 2015 openweather requires that you provide an APPID
with your HTTP requests. You can get on by creating a
free account at: http://openweathermap.org/appid

To Test: Use browser to view http://localhost:3000/
*/

let http = require('http')
let url = require('url')
let qstring = require('querystring')

const PORT = process.env.PORT || 3000
//Please register for your own key replace this with your own.
const API_KEY = 'f45b6d23576028c0609371dd5060e010'

function sendResponse(weatherData, res){
  var page = '<html><head><title>API Example</title></head>' +
    '<body>' +
    '<form method="post">' +
    'City: <input name="city"><br>' +
    '<input type="submit" value="Get Weather">' +
    '</form>'
  if(weatherData){
```

```
    page += '<h1>Weather Info for ' + JSON.parse(weatherData).name + '</h1><p>' +
weatherData +'</p>'
  }
  page += '</body></html>'
  res.end(page);
}

function parseWeather(weatherResponse, res) {
  let weatherData = ''
  weatherResponse.on('data', function (chunk) {
    weatherData += chunk
  })
  weatherResponse.on('end', function () {
    sendResponse(weatherData, res)
  })
}

function getWeather(city, res){

//New as of 2015: you need to provide an appid with your request.
//Many API services now require that clients register for an app id.

//Make an HTTP GET request to the openweathermap API
  let options = {
    host: 'api.openweathermap.org',
    path: '/data/2.5/weather?q=' + city +
    '&appid=' + API_KEY
  }
  http.request(options, function(apiResponse){
    parseWeather(apiResponse, res)
  }).end()
}

http.createServer(function (req, res) {
  let requestURL = req.url
  let query = url.parse(requestURL).query //GET method query parameters if any
  let method = req.method
  console.log(`${method}: ${requestURL}`)
  console.log(`query: ${query}`) //GET method query parameters if any

  if (req.method == "POST"){
    let reqData = ''
    req.on('data', function (chunk) {
      reqData += chunk
    })
    req.on('end', function() {
      console.log(reqData);
      let queryParams = qstring.parse(reqData)
```

```
      console.log(queryParams)
      getWeather(queryParams.city, res)
    })
  }
  else if (req.method == "GET"){
    let queryParams = qstring.parse(query)
    console.log(queryParams)
    if(queryParams.city) getWeather(queryParams.city, res)
    else sendResponse(null, res)

  }else{
    sendResponse(null, res)
  }
}).listen(PORT, (error) => {
  if (error)
    return console.log(error)
  console.log(`Server is listening on PORT ${PORT} CNTL-C to quit`)
})
```

WHAT IS EXPRESS JS

Express 3.x is a light-weight web application framework to help organize your web application into an MVC architecture on the server side. You can use a variety of choices for your templating language (like EJS, Jade, and Dust.js). You can then use a database like MongoDB with Mongoose (for modeling) to provide a backend for your Node.js application. Express.js basically helps you manage everything, from routes, to handling requests and views.

PURPOSE OF IT WITH NODEJS

That you don't have to repeat same code over and over again. Node.js is a low-level I/O mechanism which has an HTTP module. If you just use an HTTP module, a lot of work like parsing the payload, cookies, storing sessions (in memory or in Redis), selecting the right route pattern based on regular expressions will have to be re-implemented. With Express.js it there for you to use.

Simple Express app

```
let http              = require('http');
let url               = require('url');
var path              = require('path');
var express           = require('express');
var app               = express();

// home landing page render
```

```
router.get('/', function(req, res){
    res.render('index');
});
// Set port
app.set('port', process.env.PORT || 3000);

// Start server
app.listen(app.get('port'), function(){
  console.log("Express started on http://localhost:" + app.get('port') + ' press
Ctrl+C to terminate');
});
```

It just open the index page.

TEMPLATE RENDERING

A template engine enables you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. This approach makes it easier to design an HTML page

Another definition
A template engine is a library or a framework that uses some rules/languages to interpret data and render views. In the case of web applications, views are HTML pages (or parts of them), but they can be JSON or XML files, or, in desktop programs, GUIs. For those of you familiar with the model–view–controller concept, templates belong to the view.

DIFFERENCE BETWEEN JADE AND HANDLEBARS

- One of the main differences between Jade and Handlebars is that the former allows pretty much any JavaScript in its code whereas the latter restricts programmers to only a handful of built-in and custom-registered helpers.
- Unlike Jade, by design, Handlebars was made so that developers can't write a lot of JavaScript logic inside the templates. This helps to keep templates lean and related strictly to the representation of the data (no business logic).
- Jade is a Node.js brother of Haml, in the sense that it uses whitespace and indentation as part of its language. Therefore, we need to be careful to follow the proper syntax
- Another drastic difference between Jade and Handlebars is that the latter requires full HTML code (<, >, and so on), and for this reason it could care less about whitespace and indentation

USING JSON DATA (JSON.stringify() and JSON.parse())

JSON: JavaScript Object Notation. JSON is a syntax for storing and exchanging data. JSON is text, written with JavaScript object notation (It is a string representation of JavaScript data objects.).

Data objects because JSON does not allow functions, or Dates for example.

Node, and Browsers provide a JSON global function that can turn a JavaScript object into a JSON string:

```
var jsonString = JSON.stringify(obj);
```

and a parser to turn a JSON string back into an object:

```
var obj = JSON.parse(jsonString);
```

JavaScript vs JSON

```
var jsonString = JSON.stringify(obj);
// '{"name":"Lou","date":"2015-08-07T22:49:09.047Z"}'

var obj2 = JSON.parse(jsonString);
// { name: 'Lou', date: '2015-08-07T22:49:09.047Z' }
```

Observation

- When you JSON.stringify() and object. Functions are ignored, dates are turned into strings. Also, JSON strings only allow double quotes for key names (not single quotes).

- When you JSON.stringify() an object, functions are ignored, dates are turned into strings. Also, JSON strings only allow double quotes for keys, not single or double quotes the way JavaScript allows.

- There is no such thing as a JSON object parse. JSON is a string representation of objects.

- JSON is easy to move or store because it is just text and easy to parse with JavaScript's built in JSON function object.

Exchanging Data

- When exchanging data between a browser and a server, the data can only be text.

- JSON is text, and we can convert any JavaScript object into JSON, and send JSON to the server.

- We can also convert any JSON received from the server into JavaScript objects.

- This way we can work with the data as JavaScript objects, with no complicated parsing and translations.

Sending Data

If you have data stored in a JavaScript object, you can convert the object into JSON, and send it to a server:

```
var myObj = { "name":"John", "age":31, "city":"New York" };
var myJSON = JSON.stringify(myObj);
window.location = "demo_json.php?x=" + myJSON;
```

Receiving Data

If you receive data in JSON format, you can convert it into a JavaScript object:

```
var myJSON = '{ "name":"John", "age":31, "city":"New York" }';
var myObj = JSON.parse(myJSON);
document.getElementById("demo").innerHTML = myObj.name;
```

JSON Syntax Rules

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

```
// JSON data
{ "name":"John" }

// In the JavaScript, it will become
{ name:"John" }
```

JSON Values

In JSON, values must be one of the following data types:

- a string
- a number
- an object (JSON object)
- an array

- a Boolean
- null

JSON values cannot be one of the following data types:

- a function
- a date
- undefined

JSON Arrays

The same way JavaScript objects can be used as JSON, JavaScript arrays can also be used as JSON.

```
{"employees":[
    { "firstName":"John", "lastName":"Doe" },
    { "firstName":"Anna", "lastName":"Smith" },
    { "firstName":"Peter", "lastName":"Jones" }
]}
```

JSON Files

- The file type for JSON files is ".json"
- The MIME type for JSON text is "application/json"

EXPRESS ROUTES AND MIDDLEWARE

The **routes** subdirectory contains the web endpoint applications that listen for web requests and render web pages.

Simple express app

```
var express = require('express');
var app = express();
app.get('/', function (req, res) {
    res.send('Hello World!');
});
app.listen(3000, function () {
    console.log('Example app listening on port 3000!');
});
```

Example of express folder style

```
├─── app.js
├─── bin
|      └─── www
├─── package.json
├─── public
|      ├─── images
|      ├─── javascripts
|      └─── stylesheets
|      └─── style.css
├─── routes
|      ├─── index.js
|      └─── users.js
└─── views
       ├─── error.jade
       ├─── index.jade
       └─── layout.jade
```

If you do not want to create this folder and files you can user express generator with following code and it will create this structure for you.

```
sudo npm install express-generator -g
express bookapp
npm install
DEBUG=bookapp:* npm start
```

For using routes you have to define routes in your app.js. For example:

```
var routes            = require('./routes/index');
```

this command provides your route JavaScript file and you can send them notification with following code

```
app.use('/', routes);
```

in the routes you have to exports the router in the end of the page

```
module.exports = router;
```

You also have to declere express and router again in the top of the page

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

An example for rendering a page

```
// index render
router.get('/recipe', ensureAuthenticated, function(req, res){
  Recipe.find().then(function(doc){
    res.render('index', { items: doc, title: 'FoodX | Home Page', homeMenuClass: 
'active', aboutMenuClass: '', username: req.user.username});
  })
});
```

We are using get method because we'll take the link from browser.

The following code is an example of routes that are defined for the GET and the POST methods to the root of the app.

```
// GET method route
app.get('/', function (req, res) {
    res.send('GET request to the homepage')
})

// POST method route
app.post('/', function (req, res) {
    res.send('POST request to the homepage')
})
```

This route path will match abcd, abxcd, abRANDOMcd, ab123cd, and so on.

```
app.get('/ab*cd', function (req, res) {
    res.send('ab*cd')
})
```

To define routes with route parameters, simply specify the route parameters in the path of the route as shown below.

```
app.get('/users/:userId/books/:bookId', function (req, res) {
    res.send(req.params)
})
```

More than one callback function can handle a route (make sure you specify the next object). For example:

```
app.get('/example/b', function (req, res, next) {
    console.log('the response will be sent by the next function ...')
    next()
}, function (req, res) {
    res.send('Hello from B!')
})
```

MIDDLEWARE

Middleware is the intermediary between the system/operating system/database and the application. With Express, the middleware is part of a chain of applications, each of which does a certain function related to an HTTP request — either processing it, or performing some manipulation on the request for future middleware applications. The set of middleware that works with Express is quite comprehensive.

The order in which the middleware is mounted is important so if you add additional middleware functionality, be sure that it is in relation to other middleware according to the developer recommendations.

A middleware function typically looks like the following:

```
function(req, res, next){
    //do some stuff
    next(); //allow next middleware to run
}
```

The middleware function usually calls next() as its last statement to allow the next attached middleware to run. A middleware function could end the chain by sending a response to the client and not calling next() as well but here we want to call next() because we are just logging information and don't want to interupt the normal flow and routing of the client request.

EXPRESS STATIC SERVER

Middleware is attached to the app using the .use method of the app object. The order is very important -it needs to go before the static server middleware because it needs to run before that. Middleware will be executed in the order of the .use statements.

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

const PORT = process.env.PORT || 3000
const ROOT_DIR = '/public'; //root directory for our static pages

//Middleware
app.use(function(req, res, next){
  console.log('-------------------------------');
  console.log('req.path: ', req.path);
  console.log('serving:' + __dirname + ROOT_DIR + req.path);
  next(); //allow next route or middleware to run
```

```
});
app.use(express.static(__dirname + ROOT_DIR)); //provide static server

//Routes

//start server
app.listen(PORT, err => {
  if(err) console.log(err)
  else {console.log(`Server listening on port: ${PORT}`)}
})
```

app.all vs app.use
app.all() attaches to the application's router, so it's used whenever the app.router middleware is reached (which handles all the method routes... GET, POST, etc).

app.use() attaches to the application's main middleware stack, so it's used in the order specified by middleware. eg, if you put it first, it will be the first thing run. If you put it last, (after the router), it usually won't be run at all.

PART IV
Databases

MongoDB vs SQLite (JSON vs. Relational Database),

MONGO.EXE vs. MONGODB.EXE,

DATABASE NORMALIZATION,

RELATIONAL DATABASE TABLES vs. JSON STORAGE,

JSON vs. XML

MongoDB vs SQLite (JSON vs. Relational Database)

Two data interchanges formats have become very popular: XML and JSON

Two database types seem very popular with web developers: Relational and JSON/XML native.

- Relational Databases (e.g. SQLite, MySQL)
- Native JSON databases (e.g. MongoDB)
- Native XML databases (e.g. BaseX)

Relational databases like MySQL, PostgreSQL and SQLite3 represent and store data in tables and rows. They're based on a branch of algebraic set theory known as relational algebra. Meanwhile, non-relational databases like MongoDB represent data in collections of JSON documents.

Relational databases use Structured Querying Language (SQL), making them a good choice for applications that involve the management of several transactions. The structure of a relational database allows you to link information from different tables through the use of foreign keys (or indexes), which are used to uniquely identify any atomic piece of data within that table. Other tables may refer to that foreign key, so as to create a link between their data pieces and the piece pointed to by the foreign key.

If you want your application to handle a lot of complicated querying, database transactions and routine analysis of data, you'll probably want to stick with a relational database.

A non-relational database just stores data without explicit and structured mechanisms to link data from different tables (or buckets) to one another.

Advantages of JSON Database

If your data model turns out to be very complex, or if you find yourself having to de-normalize your database schema, non-relational databases like Mongo may be the best way to go. Other reasons for choosing a non-relational database include:

- The need to store serialized arrays in JSON objects

- Storing records in the same collection that have different fields or attributes

- Finding yourself de-normalizing your database schema or coding around performance and horizontal scalability issues

- Problems easily pre-defining your schema because of the nature of your data model

- MongoDB allows developers to define the application's flow entirely on the code side.

One of the biggest advantages in going with a non-relational database is that your database is not at risk for SQL injection attacks, because non-relational databases don't use SQL and are, for the most part, schema-less. Another major advantage, at least with Mongo, is that you can theoretically shard it forever (although that does bring up replication issues).

Non-relational database disadvantages

There are no joins like there would be in relational databases. This means you need to perform multiple queries and join the data manually within your code -- and that can get very ugly, very fast.

Advantages of Relation Database

- Represents database as collection of sets, or tables, linked through keys (not pointers!!)
- User would not be concerned with storage structure
- Queries expressing in high level language (SQL –became the most important)
- Users avoid navigating through the database
- Based on very simple models of sets and relations
- **No pointers**

Relational Model -sample SQL query

SELECT grade FROM GRADES WHERE StdNo = 1223 AND Section = 100

- We are asking the database to:
    - examine all tuples (rows) in table GRADES
    - pick those satisfying some criteria
    - produce a table on the grade field of tuples that satisfy the criteria
- Notice we are not navigating the structure of the data -left up to system to do it efficiently
- There is not implicit understanding of the order in which rows will be visited!

TRADITIONAL ASSUMPTIONS ABOUT DATABASE

- There is too much data to fit in main memory.
- A Linear search of ALL the data may not be practical.
- There must only be ONE COPY of any fact in the database.
- It takes longer to locate and fetch the data than to process it.
- Disks store data in blocks and can only be accessed in units of blocks.
- There is no useful order to how the data is stored.
- Data tables, like sets, cannot have duplicates.
- Integrity is more important than speed (a corrupted database is useless).

Relation Model is based on Mathematical Sets

Important properties of Sets:

{1, 67, 23, 48}

{ x I x is red }

Elements of sets are not ordered.

Sets cannot contain duplicates.

A Set cannot contain itself

i.e {x I x is a set } cannot exist

Relations are Unordered

ACCOUNTS

	customer_name	account_no	balance
t1	Lou	1001	1000
t2	Dan	1121	500
t3	Sue	1854	3000
t4	Lou	1234	750

- Considered a set of rows and therefore rows are unordered
- Attributes (columns) are ordered
- There is a row physical order when stored in actual data files but no useful logical order because the database will visit the rows in whatever order it chooses.

Columns can have NULL values

Relation Database Keys

- A key is a minimal set of attributes (columns) whose tuple values must be unique (cannot have duplicates)
- The set of all minimal keys for a table are the candidate keys
- The primary key is the candidate key chosen by the table designer to serve as the principle key for a table (the underlined columns).
- No primary key, or part of a primary key, can be null
- Be aware SQLite allows this unless you explicitly diss-allow NULL values in primary key fields

```
CREATE TABLE employee(
  sin INTEGER PRIMARY KEY NOT NULL,
  dno INTEGER NOT NULL,
  address VARCHAR(20)
  );
```

Difference between DBMS and RDBMS

Although DBMS and RDBMS both are used to store information in physical database but there are some remarkable differences between them.

The main differences between DBMS and RDBMS are given below:

No.	DBMS	RDBMS
1)	DBMS applications store **data as file**.	RDBMS applications store **data in a tabular form**.
2)	In DBMS, data is generally stored in either a hierarchical form or a navigational form.	In RDBMS, the tables have an identifier called primary key and the data values are stored in the form of tables.
3)	**Normalization is not** present in DBMS.	**Normalization is** present in RDBMS.
4)	DBMS does **not apply any security** with regards to data manipulation.	RDBMS **defines the integrity constraint** for the purpose of ACID (Atomocity, Consistency, Isolation and Durability) property.
5)	DBMS uses file system to store data, so there will be **no relation between the tables**.	in RDBMS, data values are stored in the form of tables, so a **relationship** between these data values will be stored in the form of a table as well.
6)	DBMS has to provide some uniform methods to access the stored information.	RDBMS system supports a tabular structure of the data and a relationship between them to access the stored information.
7)	DBMS **does not support distributed database**.	RDBMS **supports distributed database**.
8)	DBMS is meant to be for small organization and **deal with small data**. it supports **single user**.	RDBMS is designed to **handle large amount of data**. it supports **multiple users**.
9)	Examples of DBMS are file systems, **xml** etc.	Example of RDBMS are **mysql**, **postgre**, **sql server**, **oracle** etc.

After observing the differences between DBMS and RDBMS, you can say that RDBMS is an extension of DBMS. There are many software products in the market today who are compatible for both DBMS and RDBMS. Means today a RDBMS application is DBMS application and vice-versa.

JSON is also a natural data format for use in the application layer. JSON supports a richer and more flexible data structure than tables made up of columns and rows.

DATABASE NORMALIZATION

- An important weakness in databases is having more than one copy of the same fact
- Normalization is the process of designing data stores so that facts are not duplicated
- Most often done in relational databases by decomposing larger tables with duplicated facts in to smaller separate tables with duplicated facts

Database normalization, or simply normalization, is the process of organizing the columns (attributes) and tables (relations) of a relational database to reduce data redundancy and improve data integrity. Normalization is also the process of simplifying the design of a database so that it achieves the optimal structure composed of atomic elements.

MongoDB

MongoDB is the most popular JSON store used with Node.js and many other applications. By JSON store we mean it stores data formatted as JSON (JavaScript Object Notation) documents. MongoDB provides several components including: The MongoDB server and a command line client shell to access it.

Mongo requires a /data/db directory to hold its data. By default, it will expect it to be at the root level of the same drive that MongoDB is installed on. You can also change the path. If you don't specify your own data/db data path mongodb will expect to use c:\data\db as a default.

You can check MongoDB version with following:

```
mongod --version

mongo --version
```

Note about users: Mongo, like most databases, allow you to set up users with different access privileges. However, if you have not set up users then connecting to the database from localhost will allow you to see everything, which is what we are doing here. So don't set up users, because once you do you need to log in as a specific user.

Mongodb stores JSON objects as documents in collections. (It actually stores BSON data which is a binary version of JSON objects.)

Adding item to database

```
> var song1 = {title: "All The Things You Are", composer: "Jerome Kern"}
> var song2 ={title: "The Girl From Ipanema", composer: "Antonio-Carlos Jobim"}
> db.getCollection("Songs")
dbSongs.Songs
> db.Songs.insert(song1)
WriteResult({ "nInserted" : 1 })
> db.Songs.insert(song2)
WriteResult({ "nInserted" : 1 })
> show collections
Songs
system.indexes
> db.Songs.find()
{ "_id" : ObjectId("5456761a5c4c86d4d5a7e6d5"), "title" : "All The Things You Ar
e", "composer" : "Jerome Kern" }
{ "_id" : ObjectId("545676245c4c86d4d5a7e6d6"), "title" : "The Girl From Ipanema
", "composer" : "Antonio-Carlos Jobim" }
>
```

Notice song objects (or more correctly documents) have an "_id" field. This is created by mongodb as a unique identifier or key.

By the way if you do: db.Songs.find().pretty() it will format the output nicer.

Accessing mongoDB from node.js app

```
const MongoClient = require('mongodb').MongoClient
const DB_PATH = 'mongodb://localhost:27017/dbSongs'


MongoClient.connect(DB_PATH, function(err, db){
   if(err) console.log(`FAILED TO CONNECTED TO: ${DB_PATH}`);
   else{
      console.log(`CONNECTED TO: ${DB_PATH}`);
      db.collection("Songs", function(err, collection){
        var cursor = collection.find();
        cursor.each(function(err,document){
           console.log(document);
           if(document == null) db.close();
        });
      });
   }
});
```

Prerequisites: You must have mongodb server running. It should have a database called "dbSongs" which contains a collection named "Songs"
You must have installed the npm module: mongodb (in the dependencies section of the package.json file)

Inserting mongoDB from node.js app (Same prerequiesites)

```
const MongoClient = require('mongodb').MongoClient;
const DB_PATH = 'mongodb://localhost:27017/dbSongs'


MongoClient.connect(DB_PATH, function(err, db){
   if(err) console.log(`FAILED TO CONNECTED TO: ${DB_PATH}`);
   else{
      console.log(`CONNECTED TO: ${DB_PATH}`);
      db.collection("Songs", function(err, collection){
        collection.insertOne({title: "Happy Birthday", composer: "Anonymous"},
function(err, object){
        var cursor = collection.find();
        cursor.each(function(err,document){
           console.log(document);
           if(document == null) db.close();
           });
      });
      });
    } //else
});
```

Inserting from file

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

//NOTE: path location and name of song data file
//and output data file is hard-coded but changeable
//here in one location

var inputFilePath = "songs/sample_songs.txt";
var outputFilePath = "songs/output.txt";
const MongoClient = require('mongodb').MongoClient;
const DB_PATH = 'mongodb://localhost:27017/dbSongs'

//parsing modes
//input mode changes when an '=' is found in data file
var MODES = {
UNKNOWN : 0,
TITLE: 1,   //parsing title of song
COMPOSER: 2, //parsing composer of song
STYLE: 3,  //parsing style of song
```

```
KEY: 4,  //parsing musical key of song
N: 5,     //place holder, no parsing
SONGDATA: 6 //parsing song chord data
};

fs.readFile(inputFilePath , function(err, data) {
  if(err) {
      console.log('ERROR OPENING FILE: ' + inputFilePath);
      throw err;
  }

  console.log('PARSING FILE: ' + inputFilePath);

  var fileDataString = data.toString(); //all data from file

  var mode = MODES.UNKNOWN;  //current parsing mode
  var parseDataString = ""; //parse data for current mode
  var currentSong = {}; //current songs being constructed
  var songsArray = []; //array of parsed songs

  function isEmptyObject(anObject){
     //answer whether anObject is empty
     for(var item in anObject)
        if(anObject.hasOwnProperty(item)) return false;
     return true;
  }


  function setMode(newMode){


      //now leaving mode
      if(mode === MODES.TITLE){
           currentSong.title = parseDataString;
      }
      else if(mode === MODES.COMPOSER) {
           currentSong.composer = parseDataString;
      }
      else if(mode === MODES.STYLE) {
           currentSong.style = parseDataString;
      }
      else if(mode === MODES.KEY) {
           currentSong.key = parseDataString;
      }
      else if(mode === MODES.SONGDATA) {
           currentSong.songData = parseDataString;

      }
```

```
    //now entering mode
    if(newMode === MODES.TITLE) {
          if(!isEmptyObject(currentSong))
              songsArray.push(currentSong);
          currentSong = {}; //make new empty song;

    }

    mode = newMode;
    parseDataString = "";
}

//parse the file data
for(var i=0; i<fileDataString.length; i++){
   if(fileDataString.charAt(i) == "="){
     //change parsing mode
     if(mode === MODES.UNKNOWN) setMode(MODES.TITLE);
     else if(mode === MODES.TITLE) setMode(MODES.COMPOSER);
     else if(mode === MODES.COMPOSER) setMode(MODES.STYLE);
     else if(mode === MODES.STYLE) setMode(MODES.KEY);
     else if(mode === MODES.KEY) setMode(MODES.N);
     else if(mode === MODES.N) setMode(MODES.SONGDATA);
     else if(mode === MODES.SONGDATA) setMode(MODES.TITLE);
   }
   else{
     //add data character to content for mode
     parseDataString = parseDataString + fileDataString.charAt(i);
   }

} //end parse data file

//write parsed songs to console
console.log(songsArray);

var dataAsObject = {};
dataAsObject.songs = songsArray;

//write parsed songs to output file.
//write the array as a stringified JSON object.

fs.writeFile(outputFilePath , JSON.stringify(dataAsObject, null, 2), function(err){
  if(err) console.log(err);
  else console.log('file was saved to: ' + outputFilePath);
});

 //write parsed songs to mongo datase.
 MongoClient.connect(DB_PATH, function(err, db){
```

```
      if(err) console.log(`FAILED TO CONNECTED TO: ${DB_PATH}`);
      else{
         console.log(`CONNECTED TO: ${DB_PATH}`);
         db.collection("Songs", function(err, collection){
        for(let i=0; i<dataAsObject.songs.length; i++){
              collection.insertOne(dataAsObject.songs[i], function(err, object){});
          }
      db.close();
         });
      } //else
   });

});
```

MONGO.EXE vs. MONGODB.EXE

The mongod.exe which runs the database server itself and mongo.exe which is a command-line client that can connect to the server. Mongo.exe server is listening for connections on port 27017. This is its default port.

RELATIONAL DATABASE TABLES vs. JSON STORAGE

A relational database structures data into tables and rows while MongoDB structures data into collections of JSON documents

JSON vs. XML

Both JSON and XML can be used to receive data from a web server. The following JSON and XML examples both defines an employees object, with an array of 3 employees:

```
// JSON
{"employees":[
  { "firstName":"John", "lastName":"Doe" },
  { "firstName":"Anna", "lastName":"Smith" },
  { "firstName":"Peter", "lastName":"Jones" }
]}


// XML
<employees>
    <employee>
        <firstName>John</firstName> <lastName>Doe</lastName>
    </employee>
    <employee>
        <firstName>Anna</firstName> <lastName>Smith</lastName>
    </employee>
    <employee>
        <firstName>Peter</firstName> <lastName>Jones</lastName>
    </employee>
</employees>
```

JSON is Like XML Because

- Both JSON and XML are "self describing" (human readable)
- Both JSON and XML are hierarchical (values within values)
- Both JSON and XML can be parsed and used by lots of programming languages
- Both JSON and XML can be fetched with an XMLHttpRequest

JSON is Unlike XML Because

- JSON doesn't use end tag
- JSON is shorter (less verbose)
- JSON is quicker to read and write
- JSON can use arrays

Why JSON is Better Than XML

XML is much more difficult to parse than JSON. JSON is parsed into a ready-to-use JavaScript object. For AJAX applications, JSON is faster and easier than XML