



# NODE.JS

banuprakashc@yahoo.co.in



# What's Node.js

---

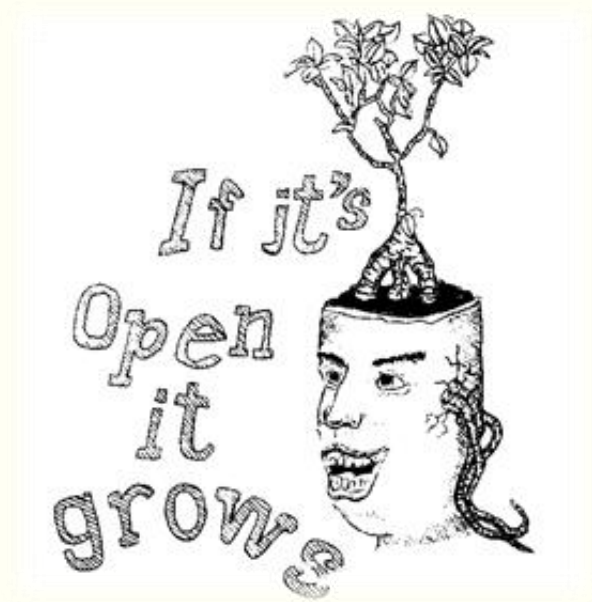
- Node.js is a platform built on Chrome's JavaScript [Google V8 Engine] runtime for easily building fast, scalable network applications.
- Node.js uses an **event-driven, non-blocking I/O model** that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.
- Server side javascript framework
- Uses CommonJS module system
- Has the latest and greatest EcmaScript5 features
- Created by **Ray Daul** in 2009



# What's Node.js

---

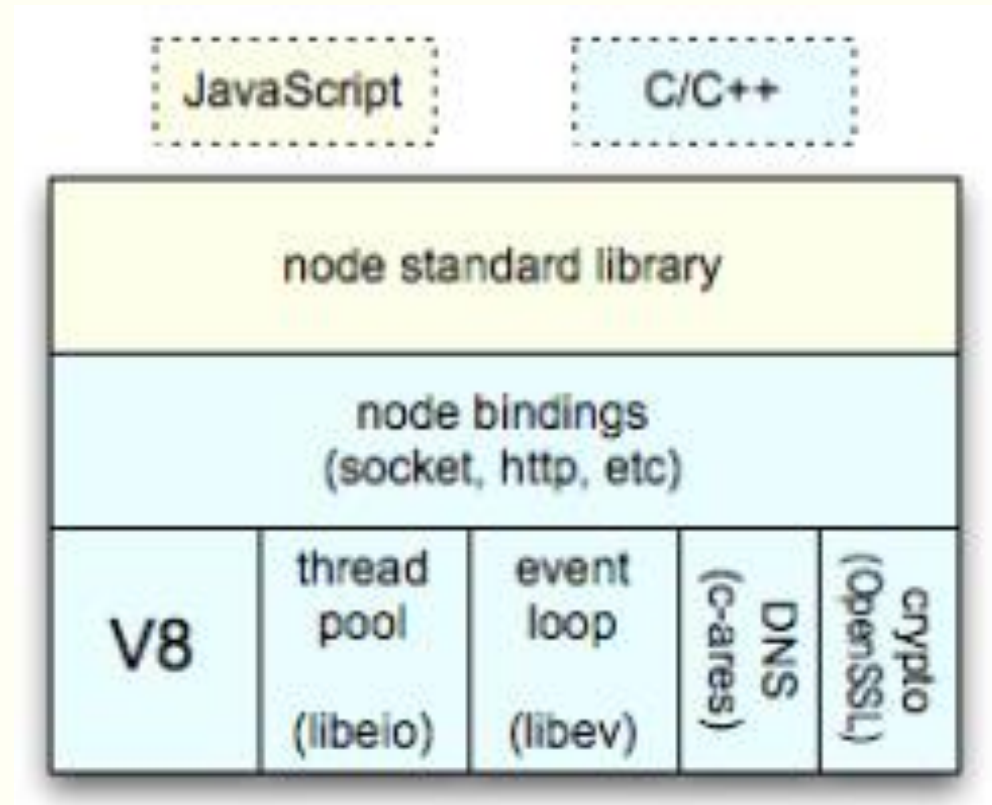
- It's Open Source



# Architecture

---

- libev(event loop)
- libeio(non-block thread pool)
- V8(Javascript engine)



# Why JavaScript?

---

- JavaScript used in client-side but node.js puts the JavaScript on server-side thus making communication between client and server will happen in same language



# Why Node.js?

---

*Build Fast!*



- Less keystrokes with JS
- JS on server and client allows for more code reuse
- A lite stack (quick create-test cycle)
- Large number of offerings for web app creation

# Why Node.js?

---

*Run Fast!*



- Fast V8 Engine
- Great I/O performance with event loop!
- Small number of layers
- Horizontal Scaling

# Why Node.js?

---

*Adapt Fast!*



- JS across stack allows easier refactoring
- Smaller codebase
- See #1 (Build Fast!)



# Why Node.js?

---

- Developers can write web applications in one language, which helps by reducing the context switch between client and server development, and allowing for code sharing between client and server.
- JSON is a very popular data interchange format today and is native to JavaScript
- JavaScript is the language used in various NoSQL databases (such as CouchDB and MongoDB), so interfacing with them is a natural fit (for example, MongoDB's shell and query language is JavaScript; CouchDB's map/reduce is JavaScript).
- Node uses one virtual machine (V8) that keeps up with the ECMAScript standard.
- In other words, you don't have to wait for all the browsers to catch up to use new JavaScript language features in Node.

# Why Node.js?

---

- Node provides an event-driven and asynchronous platform for server-side JavaScript.
- It brings JavaScript to the server in much the same way a browser brings JavaScript to the client.
- Both are event-driven (they use an event loop) and non-blocking when handling I/O (they use asynchronous I/O).

# Why Node.js?

---



# Node.js event loop

---

- The first basic thesis of node.js is that I/O is expensive:

## The cost of I/O

L1-cache	3 cycles
L2-cache	14 cycles
RAM	250 cycles
Disk	41 000 000 cycles
Network	240 000 000 cycles

- So the largest time wasted with current programming technologies comes from waiting for I/O to complete

# Node.js event loop

---

- There are several ways in which one can deal with the performance impact
  - **synchronous**: you handle one request at a time, each in turn.
    - *pros*: simple
    - *cons*: any one request can hold up all the other requests
  - **fork a new process**: you start a new process to handle each request.
    - *pros*: easy
    - *cons*: does not scale well, hundreds of connections means hundreds of processes.
    - `fork()` is the Unix programmer's hammer. Because it's available, every problem looks like a nail. It's usually overkill
  - **threads**: start a new thread to handle each request.
    - *pros*: easy, and kinder to the kernel than using `fork`, since threads usually have much less overhead
    - *cons*: your machine may not have threads, and threaded programming can get very complicated very fast, with worries about controlling access to shared resources.

# Node.js event loop

---

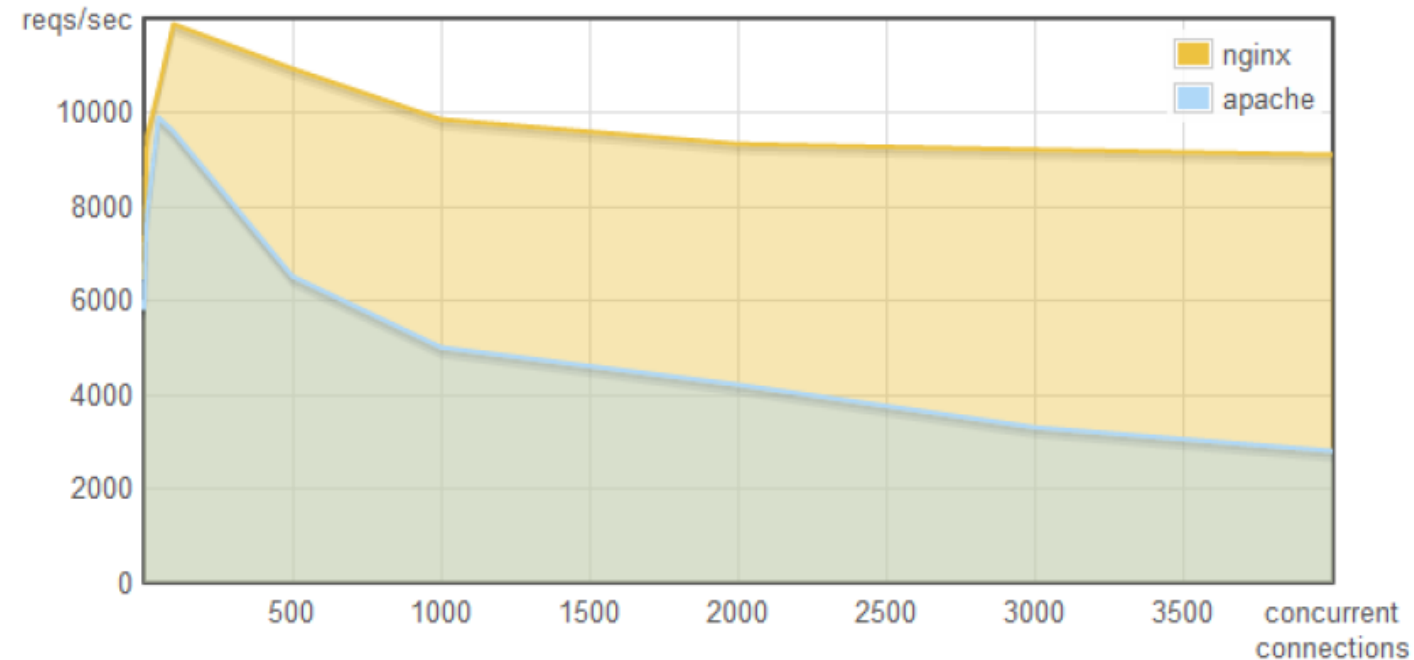
- The second basis thesis is that thread-per-connection is memory-expensive.
  - NGINX (<http://nginx.com/>), is an HTTP server like Apache, but instead of using the multithreaded approach with blocking I/O, it uses an event loop with asynchronous I/O.
  - Apache uses one thread per connection.
  - NGINX doesn't use threads. It uses an **event loop**.

# Node.js event loop

---

- **Apache vs NGINX** [concurrency x reqs/sec]

concurrency  $\times$  reqs/sec



# Node.js event loop

---

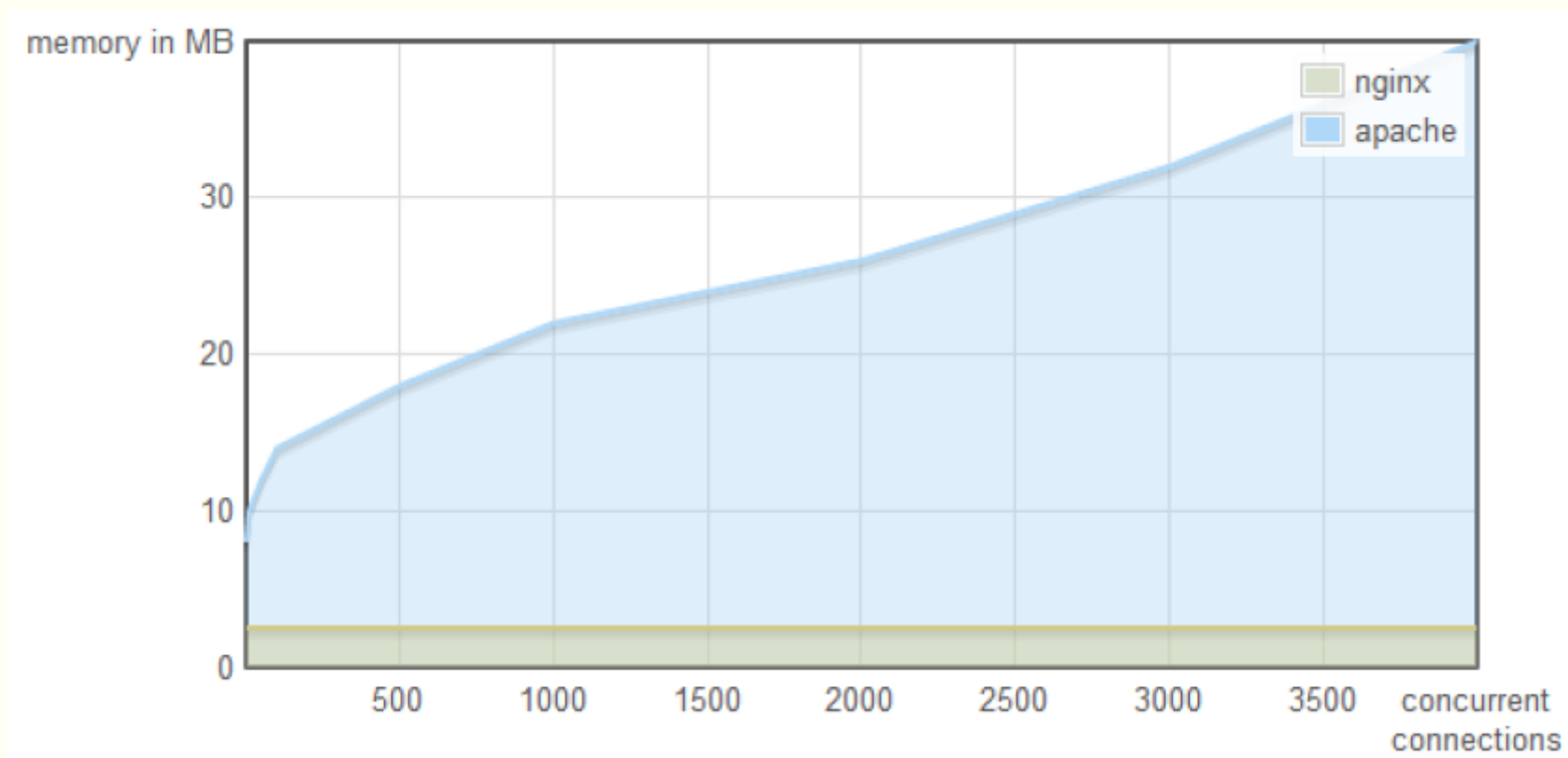
Platform	Number of request per second
PHP ( via Apache)	3187,27
Static ( via Apache )	2966,51
Node.js	5569,30



# Node.js event loop

---

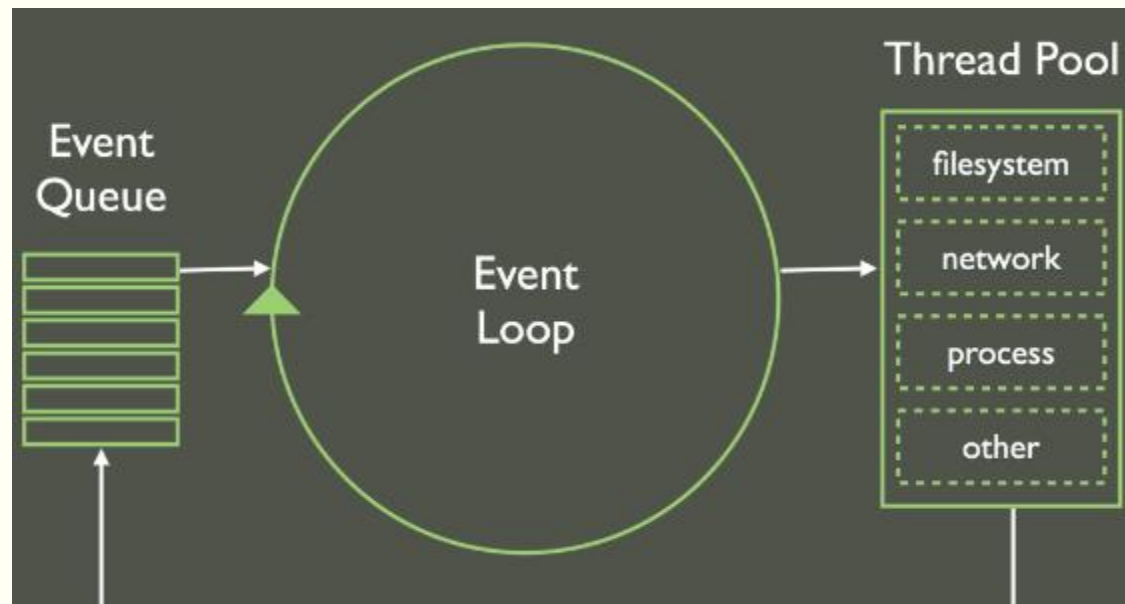
- **Apache vs NGINX** [concurrency x memory]



# Node.js event loop

---

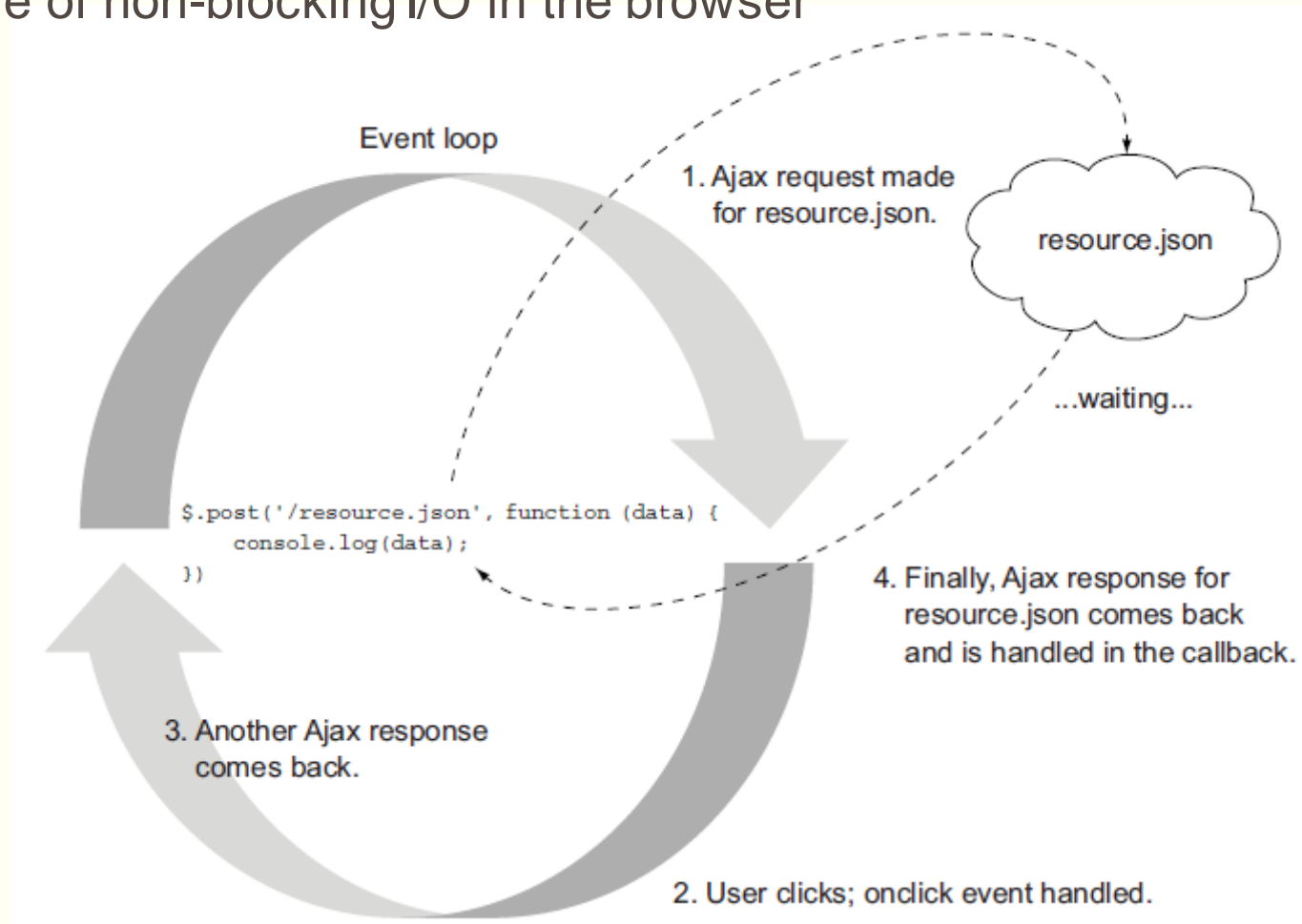
- Events will be stored in the Event Queue, each time the Event Loop comes around, it will take the top item in Event Queue, which will then be processed in the Thread Pool.
- The best part of this whole process is that the items only wait as long as it takes for the actual work to be done, as opposed for their next logical step to finish.



# Node.js event loop

---

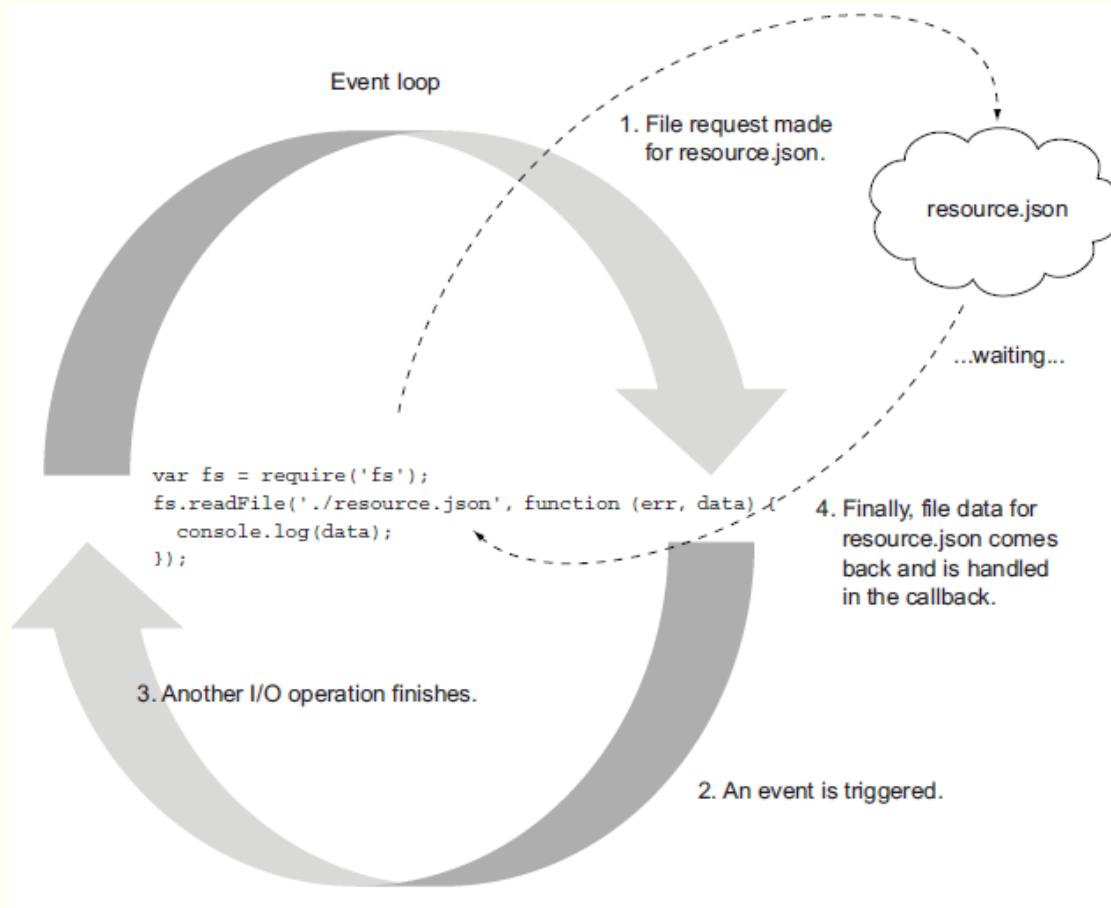
- An example of non-blocking I/O in the browser



# Node.js event loop

---

- An example of non-blocking I/O in Node



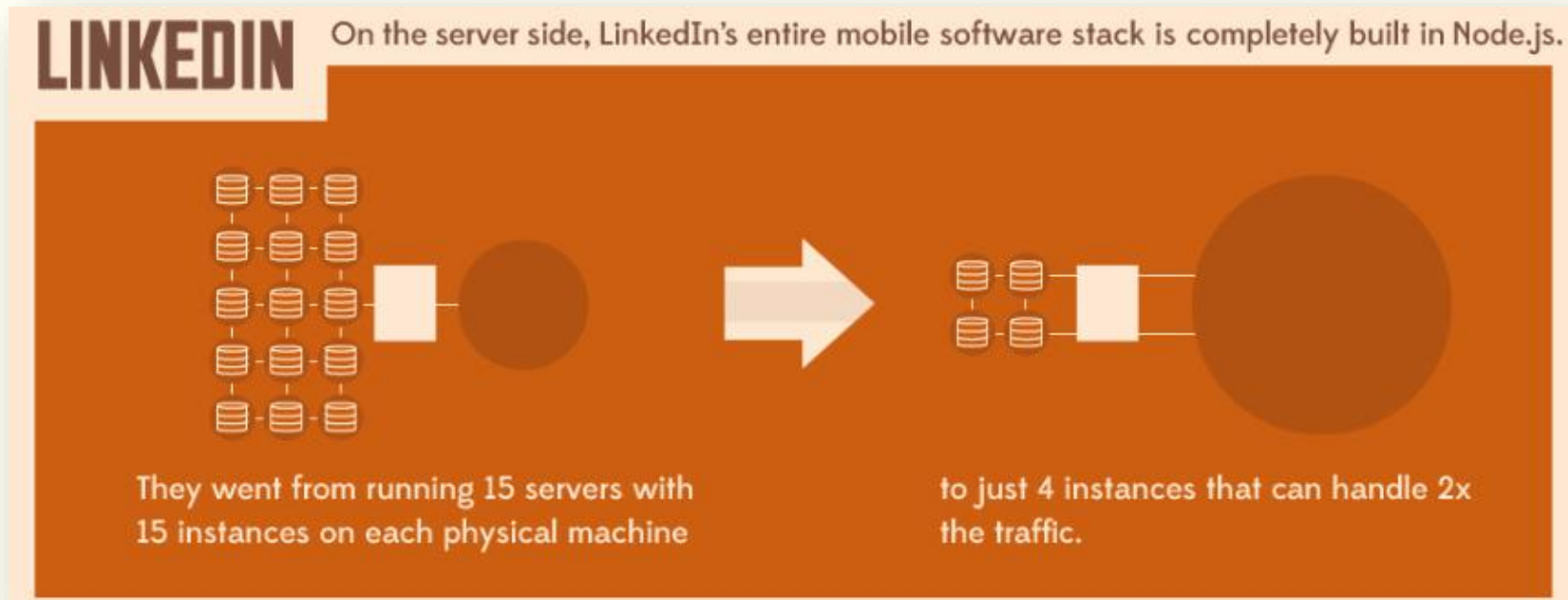
# Node Architecture

---

LinkedIn claims:

Ruby on rails to node

Node.js being up to 20x faster than Rails for certain scenarios



# Node.js Good Use Cases

---

## ▪ JSON APIs

- Building light-weight REST / JSON api's is something where node.js really shines.
- Its non-blocking I/O model combined with JavaScript make it a great choice for wrapping other data sources such as databases or web services and exposing them via a JSON interface.

## ▪ Single page apps

- If you are planning to write an AJAX heavy single page app (think gmail), node.js is a great fit as well.
- The ability to process many requests / seconds with low response times, as well as sharing things like validation code between the client and server make it a great choice for modern web applications that do lots of processing on the client

# Node.js Good Use Cases

---

- **Streaming data**

- Traditional web stacks often treat http requests and responses as atomic events. However, the truth is that they are streams, and many cool node.js applications can be built to take advantage of this fact.

- **Realtime Applications**

- Another great aspect of node.js is the ease at which you can develop soft real time systems like twitter, chat software, sport bets or interfaces to instant messaging networks

# Node.js Bad use cases

---

- **CPU heavy apps**

- Apps that are very heavy on CPU usage, and very light on actual I/O. So if you're planning to write video encoding software, artificial intelligence or similar CPU hungry software, please do not use node.js

- **Simple CRUD / HTML apps**

- If most of your app is simply rendering HTML based on some database, using node.js will not provide many tangible business benefits yet. Your app will not magically get more traffic just because you write it in node.js



# Node.js Modules

---

- Node.JS - a Common.JS Module Implementation
- Your code uses require to include modules.
- Modules use exports to make things available.

# CommonJS

---

- An ecosystem for JavaScript outside the browser
  - Modules
  - Promises
  - Binary
  - Filesystem
  - Console
  - System
  - Testing
- CommonJS Implementations
  - MongoDB
  - CouchDB
  - SproutCore
  - Node.JS

# Node Globals

---

**process** – object providing information and methods for the current process

process.stdout

process.stderr

process.stdin

process.argv

process.env

- **console** – allows printing to stdout and stderr
- **require()** – function to load a module
- **module** – refers to the current module

# The Simplest Module

---

- `// hello.js`
- `console.log('Hello World');`
- `// app.js`
- `require('hello.js');`

# Pattern 1: Define a Global

---

```
/**
 * Math Module
 * math.js
 */

add = function(first, second) {
    return first + second;
};

subtract = function(first, second) {
    return first - second;
};

/**
 * Client Module
 * client.js
 */

require('./math.js');

console.log(add(3,4));

console.log(subtract(10,2));
```

don't pollute the global space



## Pattern 2: Export an Anonymous Function

---

```
/**
 * Cart Module
 * cart.js
 */
module.exports = (function(){
    var cart = [];
    function add(item) {
        cart.push(item);
    }
    function get(){
        return cart;
    }
    return {
        add : add,
        get : get
    }
})();
```

```
/**
 * Client Module
 * cartClient.js
 */

var cart = require('./cart.js');

cart.add({"id": 3,
         "name" : 'Mobile',
         "price": 25000.00});

cart.add({"id": 4,
         "name" : 'Laptop',
         "price": 55000.00});

var items = cart.get();

console.log(items);
```

## Pattern 3: Export a Named Function

---

```
/**
 * Cart Module
 * cart.js
 */
exports.cart = (function(){
    var cart = [];
    function add(item) {
        cart.push(item);
    }
    function get(){
        return cart;
    }
    return {
        add : add,
        get : get
    }
})();
```

```
/**
 * Client Module
 * cartClient.js
 */
var cart = require('./cart.js').cart;

cart.add({"id": 3,
    "name" : 'Mobile',
    "price": 25000.00});

cart.add({"id": 4,
    "name" : 'Laptop',
    "price": 55000.00});

var items = cart.get();

console.log(items);
```

# exports VS. module.exports

---

- exports is an alias to module.exports.
- node automatically creates it as a convenient shortcut.
- For assigning named properties, use either one
  - `module.exports.fiz = "fiz";`
  - `exports.buz = "buz";`
  - `module.exports === exports;     //true`



# Warning

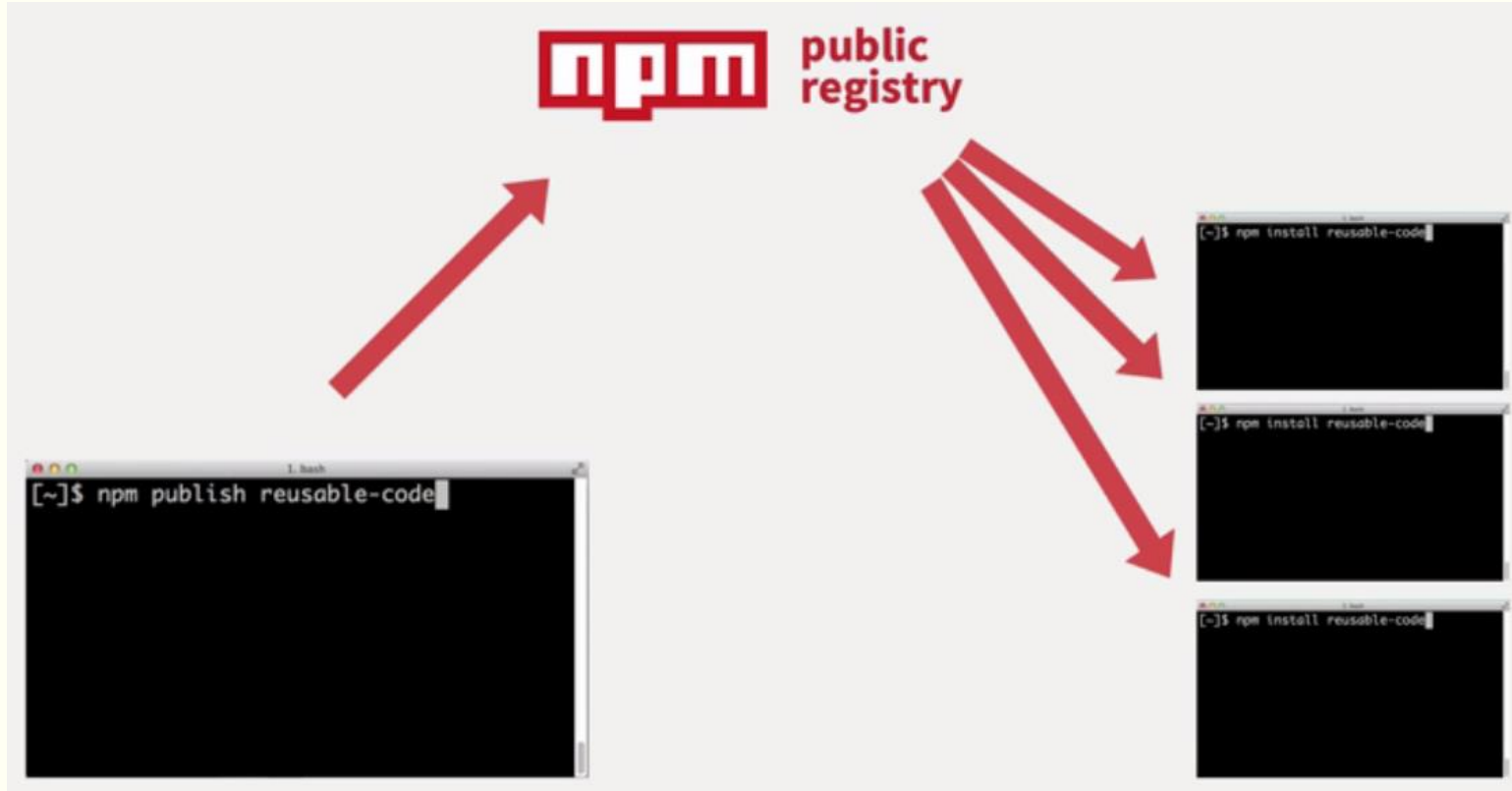
---

- Assigning anything to exports directly (instead of exports.something) will overwrite the exports alias
  - `module.exports.qux = "qux";`
  - `> exports`
    - `{ qux: "qux" }`
  - `> exports === module.exports`
    - `true`
  - `> exports = "wobble wibble wubble!";`
  - `> exports === module.exports`
    - `false`
  - `> exports`
    - `"wobble wibble wubble!"`
  - `> module.exports`
    - `{ qux: "qux" }`      `// module.exports is canonical`

# What is NPM?

---

- Node Package Manager (NPM) is a package manager for node
- Allows us to install packages from repo, and publish our own



# Node Packaged Modules

---

- Packages or Modules is just a director with one or more reusable-files in it along with a file called “package.json” in it.
- Packages are generally small and generally solve one problem



- Package.json contains metadata about the package

---

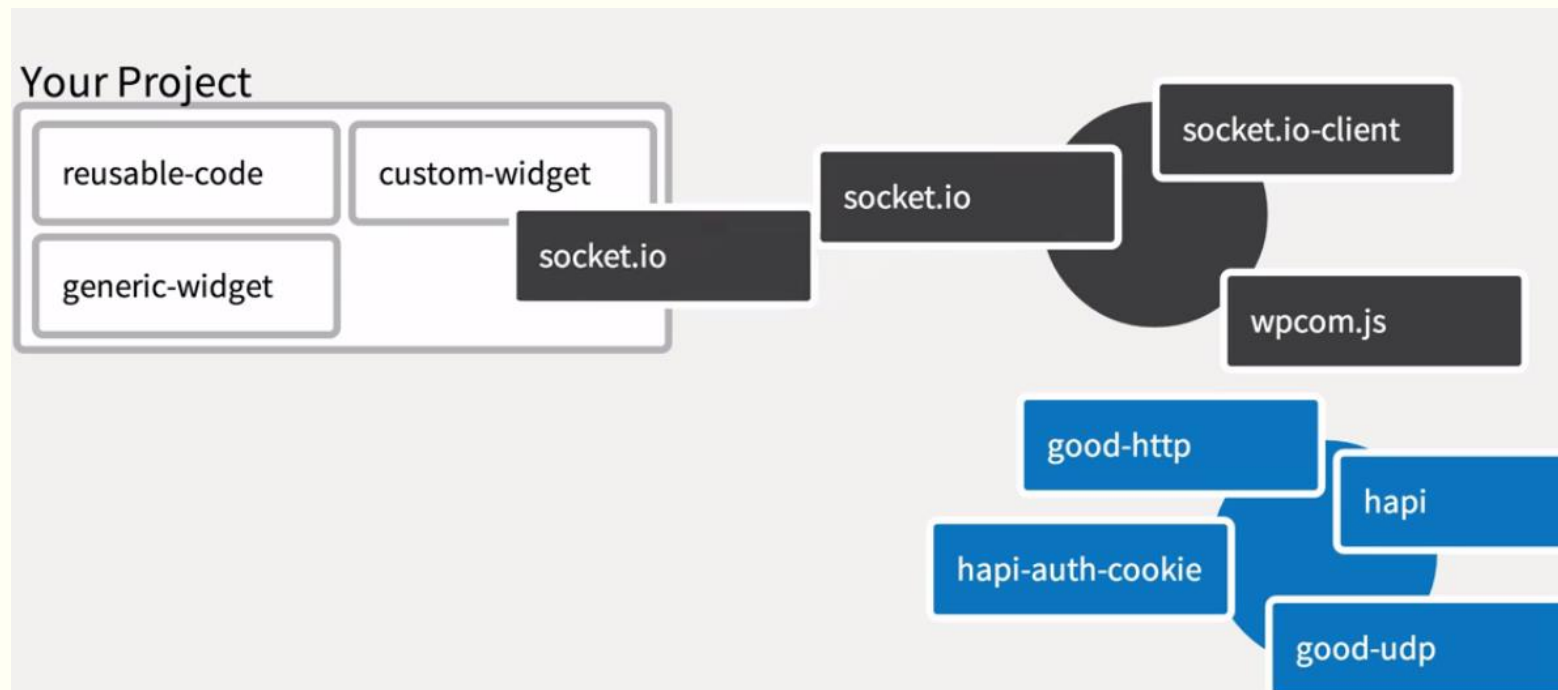
- | Table 1. Continued |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |     |
|--------------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|-----|
| 10                 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 |



# Node Packaged Modules

---

- Packages help in bringing in packages developed from people who have focussed on particular problem area



# Node Packaged Modules

---

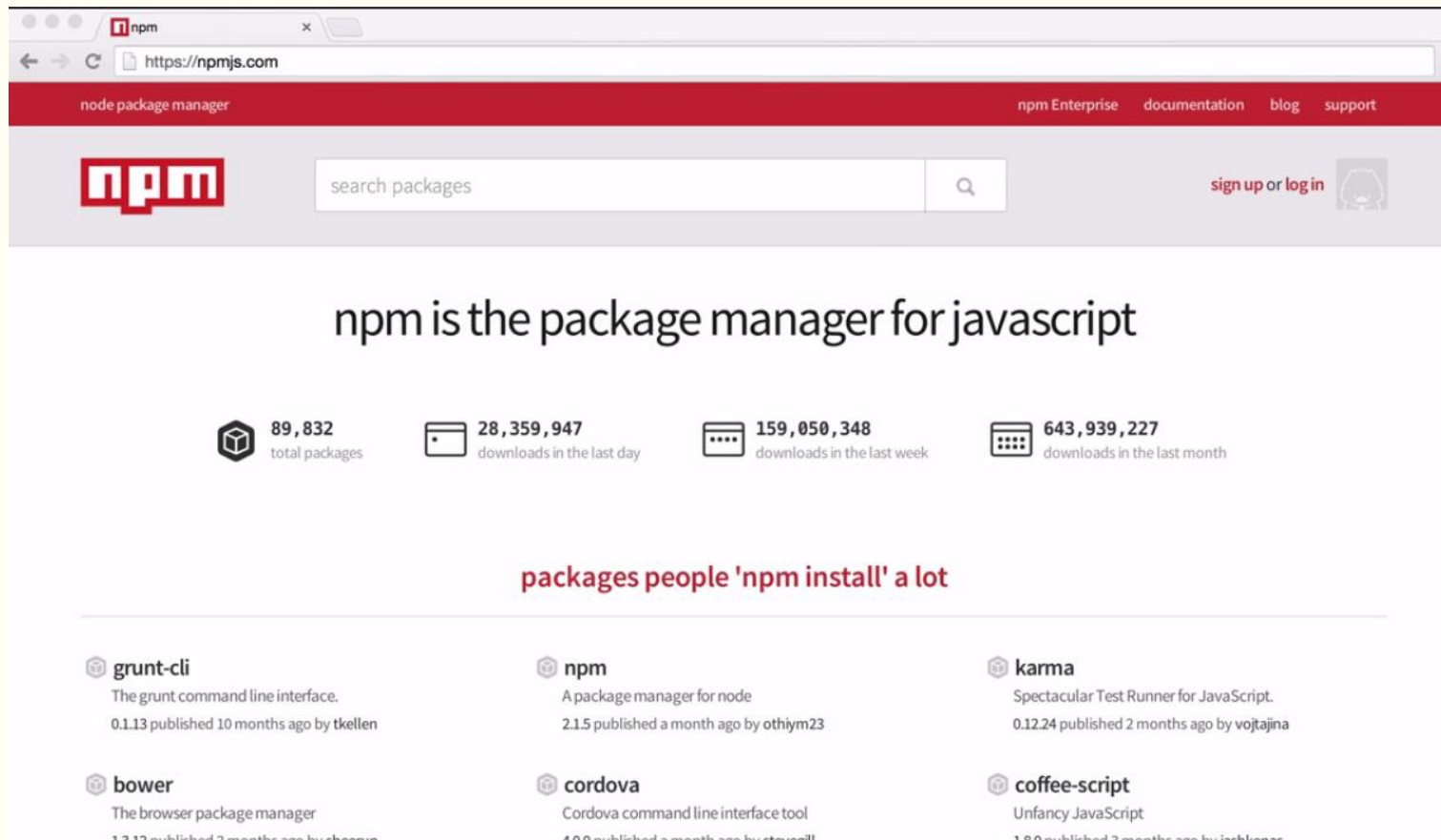
- Helps reuse code across projects



# Node Packaged Modules

---

- You can browse node module packages in npm web site



# Node Packaged Modules

---

- Node modules can be published using npm client.
- Published modules are reflected in the registry and visible in the web site. Once it is available in registry other clients can install it using npm client





# Node Packaged Modules

---

- npm can install packages in local or global mode.
- In local mode it installs the package in a `node_modules` folder in your parent working directory. This location is owned by the current user.
- Global packages are installed in `{prefix}/lib/node_modules/` which is owned by root (where `{prefix}` is usually `/usr/` or `/usr/local`).



# Changing the Location of Global Packages

---

- Let's see what output npm config gives us.

```
G:\learnyounode>npm config list
; cli configs
user-agent = "npm/2.5.1 node/v0.12.0 win32 x64"

; userconfig C:\Users\Banu Prakash\.npmrc
https-proxy = "http://172.22.218.218:8085/"
proxy = "http://172.22.218.218:8085/"

; builtin config undefined
prefix = "C:\\Users\\Banu Prakash\\AppData\\Roaming\\npm"

; node bin location = G:\nodeJS0.12\\node.exe
; cwd = G:\learnyounode
; HOME = C:\Users\Banu Prakash
; 'npm config ls -l' to show all defaults.
```

# Changing the Location of Global Packages

---

- Get the current global location

```
G:\learnnode>npm config get prefix  
C:\Users\Banu Prakash\AppData\Roaming\npm
```

- This is the prefix we want to change, so as to install global packages in our home directory. To do that create a new directory in your home folder.
  - `npm config set prefix=$HOME/.node_modules_global`
- we have altered the location to which global Node packages are installed.
- This also creates a `.npmrc` file in our home directory.
- `npm config get prefix [ /home/.node_modules_global ]`
- `cat .npmrc`
- `prefix=/home/sitepoint/.node_modules_global`

# npm install global

---

- We still have npm installed in a location owned by root. But because we changed our global package location we can take advantage of that. We need to install npm again, but this time in the new user-owned location. This will also install the latest version of npm.
  - `$ npm install npm -global`
- Finally, we need to add `.node_modules_global/bin` to our `$PATH` environment variable, so that we can run global packages from the command line.
- `export PATH="$HOME/.node_modules_global/bin:$PATH"`

# Installing Packages in Global Mode

---

```
G:\firstpack>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (firstpack) cartpack
version: (1.0.0)
description: Simple Module to add cart items
entry point: (index.js) cart.js
test command:
git repository:
keywords: cart npm module
author: banuprakashc
license: (ISC)
About to write to G:\firstpack\package.json:
```

## Installing Packages in Global Mode contd..

---

```
{
  "name": "cartpack",
  "version": "1.0.0",
  "description": "Simple Module to add cart items",
  "main": "cart.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "cart",
    "npm",
    "module"
  ],
  "author": "banuprakashc",
  "license": "ISC"
}
```

Is this ok? (yes)

# Publish the package

---

- You can publish any directory that has a package.json
- Creating a user
  - To publish, you must have a user on the npm registry. If you don't have one, create it with npm adduser.
  - If you created one on the site, use npm login to store the credentials on the client.

```
G:\firstpack>npm adduser
Username: banuprakashc
Password:
Email: (this IS public) banuprakashc@yahoo.co.in
```

# Publish the package

---

- Publishing the package

```
G:\firstpack>npm adduser
Username: banuprakashc
Password:
Email: (this IS public) banuprakashc@yahoo.co.in

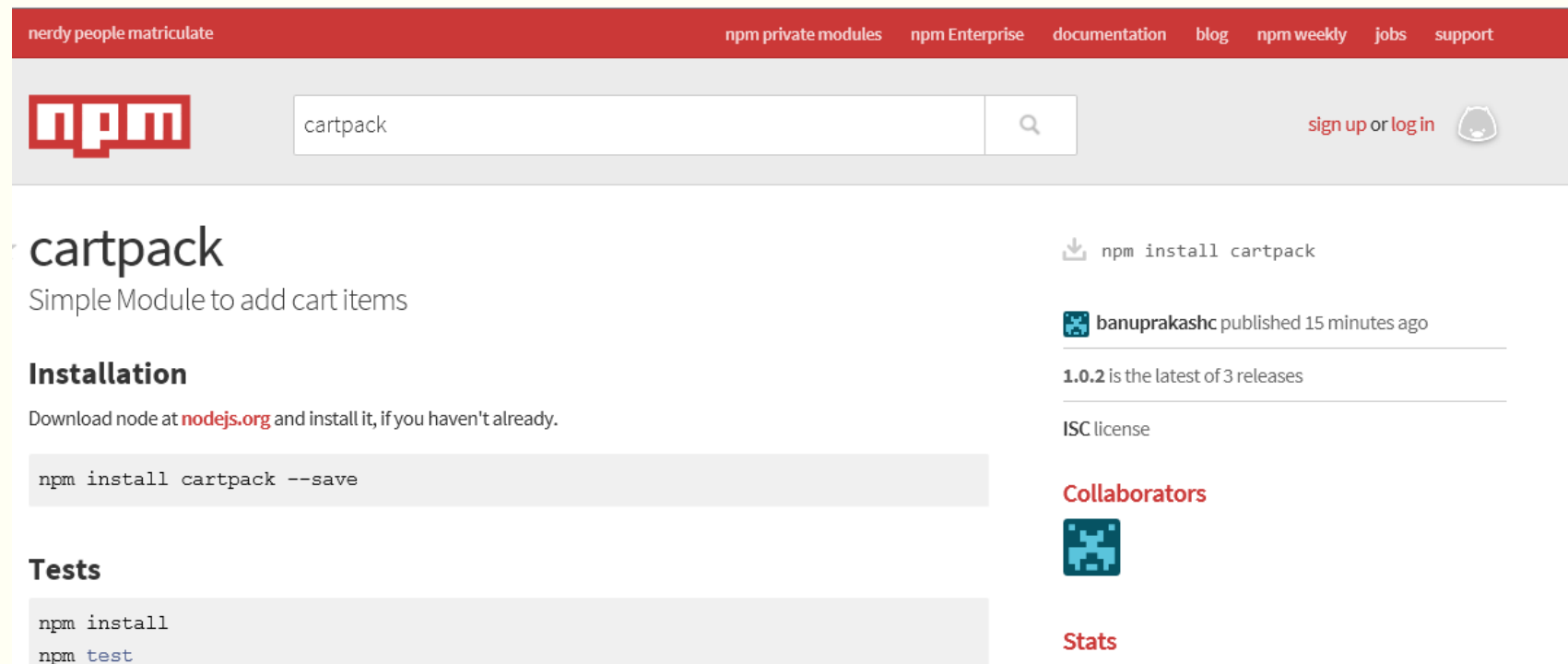
G:\firstpack>npm publish
+ cartpack@1.0.0
```

- Check published [<https://www.npmjs.com/package/cartpack>]



# View available packages

- Check published [<https://www.npmjs.com/package/cartpack>]



The screenshot shows the npm website interface. At the top, there's a red navigation bar with links like 'nerdy people matriculate', 'npm private modules', 'npm Enterprise', 'documentation', 'blog', 'npm weekly', 'jobs', and 'support'. Below this is a search bar with 'cartpack' entered and a search icon. To the right of the search bar are links for 'sign up or log in' and a GitHub icon. The main content area is divided into two columns. The left column features the package name 'cartpack' in large text, followed by the description 'Simple Module to add cart items'. Below this is an 'Installation' section with a code block containing 'npm install cartpack --save'. Further down is a 'Tests' section with a code block containing 'npm install' and 'npm test'. The right column contains a download icon and the text 'npm install cartpack', followed by the publisher's name 'banuprakashc' and the time 'published 15 minutes ago'. Below this, it states '1.0.2 is the latest of 3 releases' and 'ISC license'. At the bottom of the right column, there are sections for 'Collaborators' and 'Stats', each with a small icon.

nerdy people matriculate npm private modules npm Enterprise documentation blog npm weekly jobs support

npm

cartpack

Simple Module to add cart items

**Installation**

Download node at [nodejs.org](https://nodejs.org) and install it, if you haven't already.

```
npm install cartpack --save
```

**Tests**

```
npm install  
npm test
```

npm install cartpack

banuprakashc published 15 minutes ago

1.0.2 is the latest of 3 releases

ISC license

**Collaborators**

**Stats**

# Updating the package

---

- When you make changes, you can update the package using
- `npm version <update_type>`,
  - where `update_type` is one of the semantic versioning release types, patch, minor, or major.
  - This command will change the version number in `package.json`.
  - Note that this will also add a tag with this release number to your git repository if you have one.
- After updating the version number, you can `npm publish` again.
- Test: Go to <http://npmjs.com/package/<package>>. The package number should be updated.

# Using node modules

---

- Install module

```
G:\usepack>npm install cartpack  
cartpack@1.0.2 node_modules\cartpack
```

```
Directory of G:\usepack\node_modules\cartpack
```

```
13-05-2015  15:08    <DIR>          .  
13-05-2015  15:08    <DIR>          ..  
13-05-2015  15:08                242 cart.js  
13-05-2015  15:08                904 package.json  
13-05-2015  15:08                391 README.md  
                3 File(s)              1,537 bytes  
                2 Dir(s)  31,414,423,552 bytes free
```

# Using node modules

---

- Create test.js

```
var cart = require('cartpack').cart;

cart.add({"id": 3,
          "name" : 'Mobile',
          "price": 25000.00});

cart.add({"id": 4,
          "name" : 'Laptop',
          "price": 55000.00});

var items = cart.get();

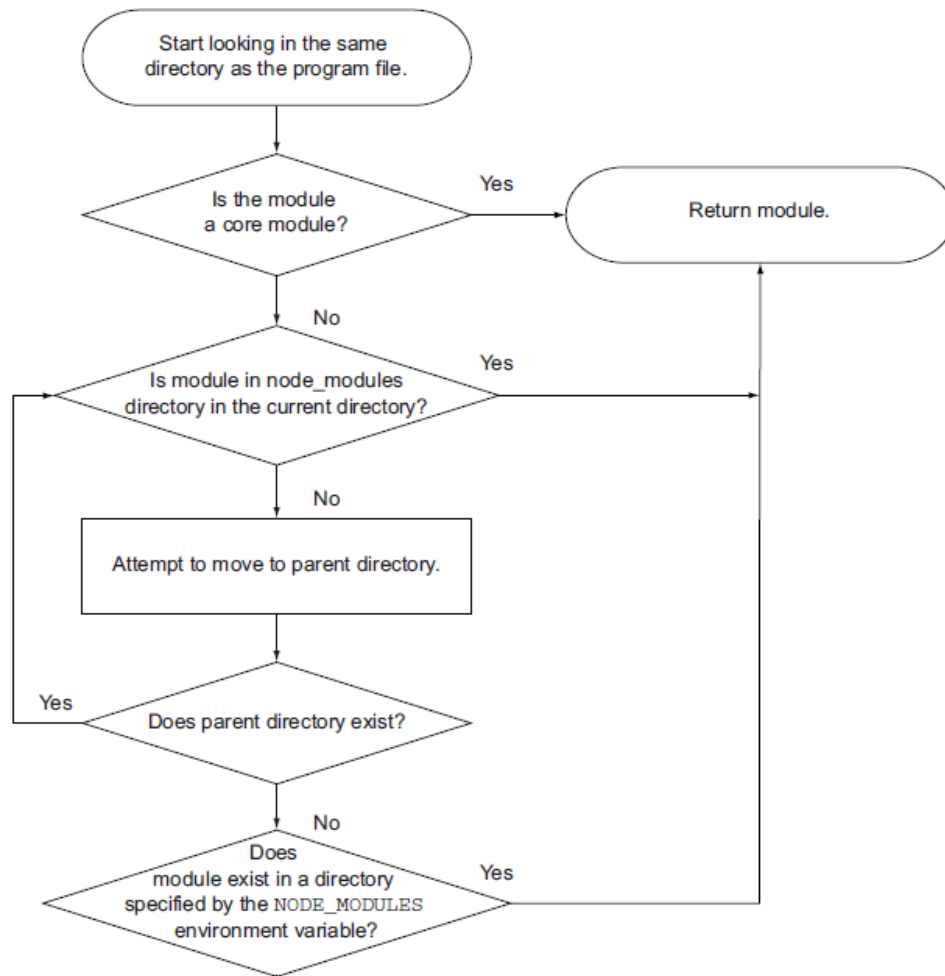
console.log(items);
```

- Invoke node

```
G:\usepack>node test.js
[ { id: 3, name: 'Mobile', price: 25000 },
  { id: 4, name: 'Laptop', price: 55000 } ]
```

# Reusing modules using the node\_modules folder

---



The `NODE_PATH` environmental variable provides a way to specify alternative locations for Node modules. If used, `NODE_PATH` should be set to a list of directories separated by semicolons in Windows or colons in other operating systems

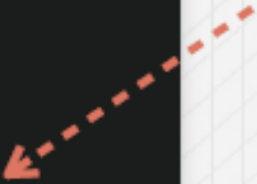
# DEFINING YOUR DEPENDENCIES

---

my\_app/package.json

```
{  
  "name": "My App",  
  "version": "1",  
  "dependencies": {  
    "connect": "1.8.7"  
  }  
}
```

*version number*



```
$ npm install
```

*Installs into the node\_modules directory*

my\_app

/ node\_modules

/ connect

# DEPENDENCIES

---

my\_app/package.json

```
"dependencies": {  
  "connect": "1.8.7"  
}
```

No conflicting modules!

Installs sub-dependencies

my\_app

/ node\_modules

/ connect

connect

/ node\_modules

/ qs

connect

/ node\_modules

/ mime

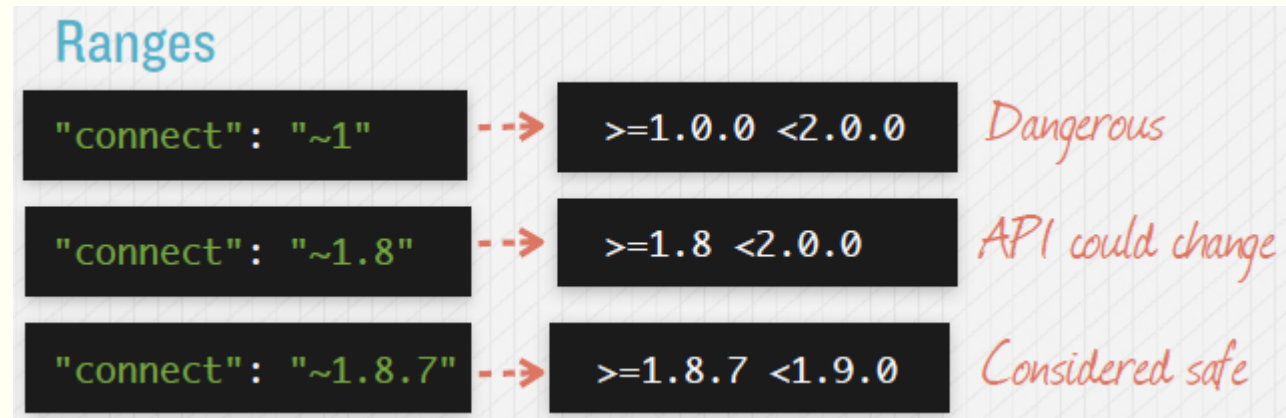
connect

/ node\_modules

/ formidable

# SEMANTIC VERSIONING

---





# Exercise

---

- **<http://nodeschool.io/index.html#workshoppers>**
  - How to npm
  - Learn how to use and create npm modules.
    - `npm install -g how-to-npm`

# File system

---

- The file system functions consist of file I/O and directory I/O functions.
- All of the file system functions offer both synchronous (blocking) and asynchronous (non-blocking) versions.
  - The difference between these two is that the synchronous functions (which have “Sync” in their name) return the value directly and prevent Node from executing any code while the I/O operation is being performed

# File System

---

```
var fs = require('fs');
var data = fs.readFileSync('./index.html', 'utf8');
// wait for the result, then use it
console.log(data);
```

Asynchronous functions return the value as a parameter to a callback given to them:

```
var fs = require('fs');
fs.readFile('./index.html', 'utf8', function(err, data) {
  // the data is passed to the callback in the second argument
  console.log(data);
});
```

# File System

---

- You should use the asynchronous version in most cases, but in rare cases (e.g. reading configuration files when starting a server) the synchronous version is more appropriate.
- Note that the asynchronous versions require a bit more thought, since the operations are started immediately and may finish in any order

```
fs.readFile('./file.html', function (err, data) {  
    // ...  
});  
fs.readFile('./other.html', function (err, data) {  
    // ..  
});
```

## Files: Reading a file

---

- Fully buffered reads and writes are fairly straightforward: call the function and pass in a String or a Buffer to write, and then check the return value

```
var fs = require("fs");
var fileName = "../helloWorld/index.html";

fs.readFile(fileName, 'utf8', function(err, data) {
  // the data is passed to the callback in the second argument
  console.log(data);
});
```

# Files: Reading a file

---

- When we want to work with files in smaller parts, we need to `open()`, get a file descriptor and then work with that file descriptor.
- `fs.open(path, flags, [mode], [callback])`
- supports the following flags:
  - `'r'` - Open file for reading. An exception occurs if the file does not exist.
  - `'r+'` - Open file for reading and writing. An exception occurs if the file does not exist.
  - `'w'` - Open file for writing. The file is created (if it does not exist) or truncated (if it exists).
  - `'w+'` - Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).
  - `'a'` - Open file for appending. The file is created if it does not exist.
  - `'a+'` - Open file for reading and appending. The file is created if it does not exist.

# Files: Reading from a file

---

- **Reading File**

- Following is the syntax of one of the methods to read from a file:

- **fs.read(fd, buffer, offset, length, position, callback)**

- Here is the description of the parameters used:

- fd - This is the file descriptor returned by file fs.open() method.
    - buffer - This is the buffer that the data will be written to.
    - offset - This is the offset in the buffer to start writing at.
    - length - This is an integer specifying the number of bytes to read.
    - position - This is an integer specifying where to begin reading from in the file. If position is null, data will be read from the current file position.
    - callback - This is the callback function which gets the three arguments, (err, bytesRead, buffer).

## Files: Reading from a file example

---

```
var fs = require("fs");
var fileName = "readExample.js";

fs.open(fileName, 'r', function(err, fd) {
  if(err) throw err;
  var buf = new Buffer(100);
  fs.read(fd, buf, 0, buf.length, null,
    function(err, bytesRead, buffer) {
      if(err) throw err;
      console.log(bytesRead, buffer.toString());
      fs.close(fd, function() {
        console.log('Done');
      });
    });
});
```



# Files: Writing to a file

---

```
var fs = require("fs");
var fileName = "test.txt";

fs.open(fileName, 'w', function(err, fd) {
  if(err) throw err;

  var buf = new Buffer('Hello World\n');

  fs.write(fd, buf, 0, buf.length, null, function(err, written, buffer) {
    if(err) throw err;

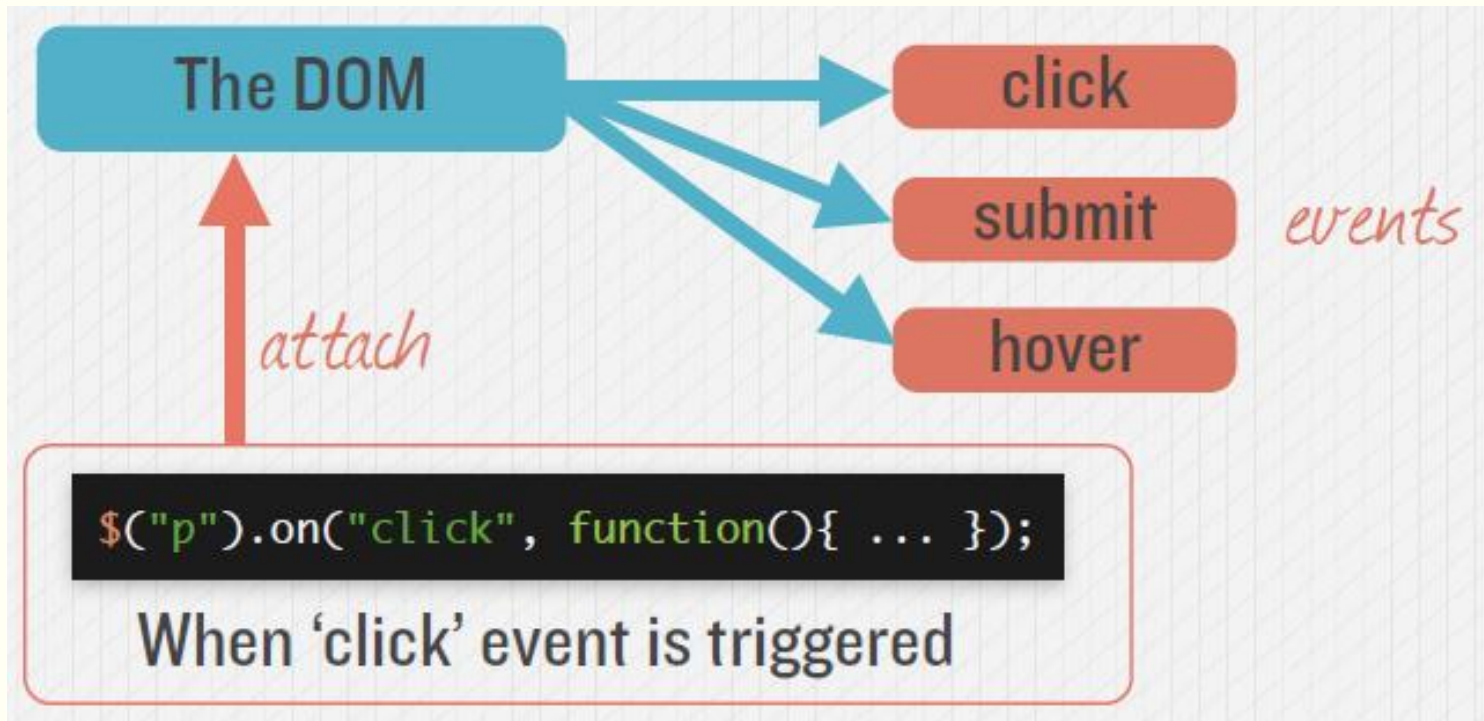
    console.log(err, written, buffer);
    //null 12 <Buffer 48 65 6c 6c 6f 20 57 6f 72 6c 64 0a>

    fs.close(fd, function() {
      console.log('Done');
    });
  });
});
```

# Events

---

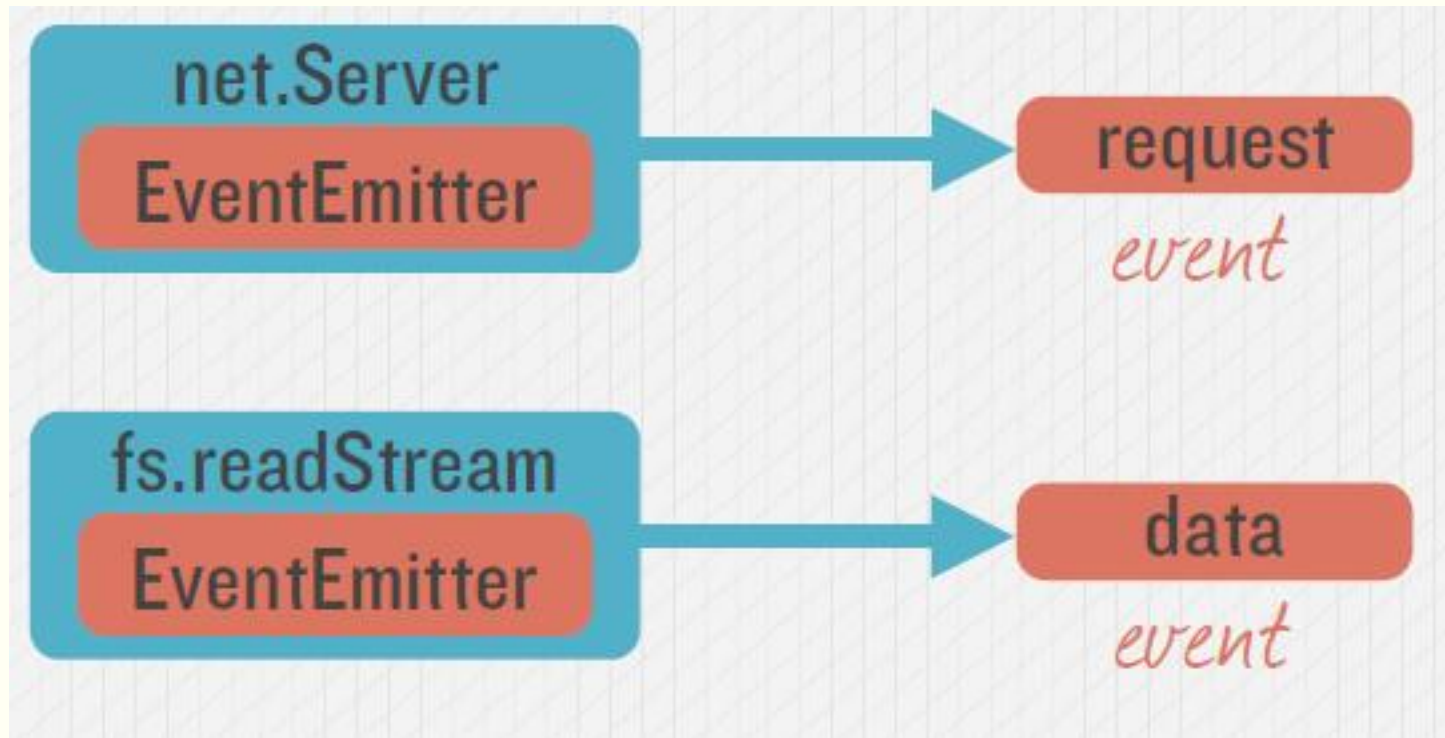
- EVENTS IN THE DOM
- The DOM triggers Events; you can listen for those events



# EVENTS IN NODE

---

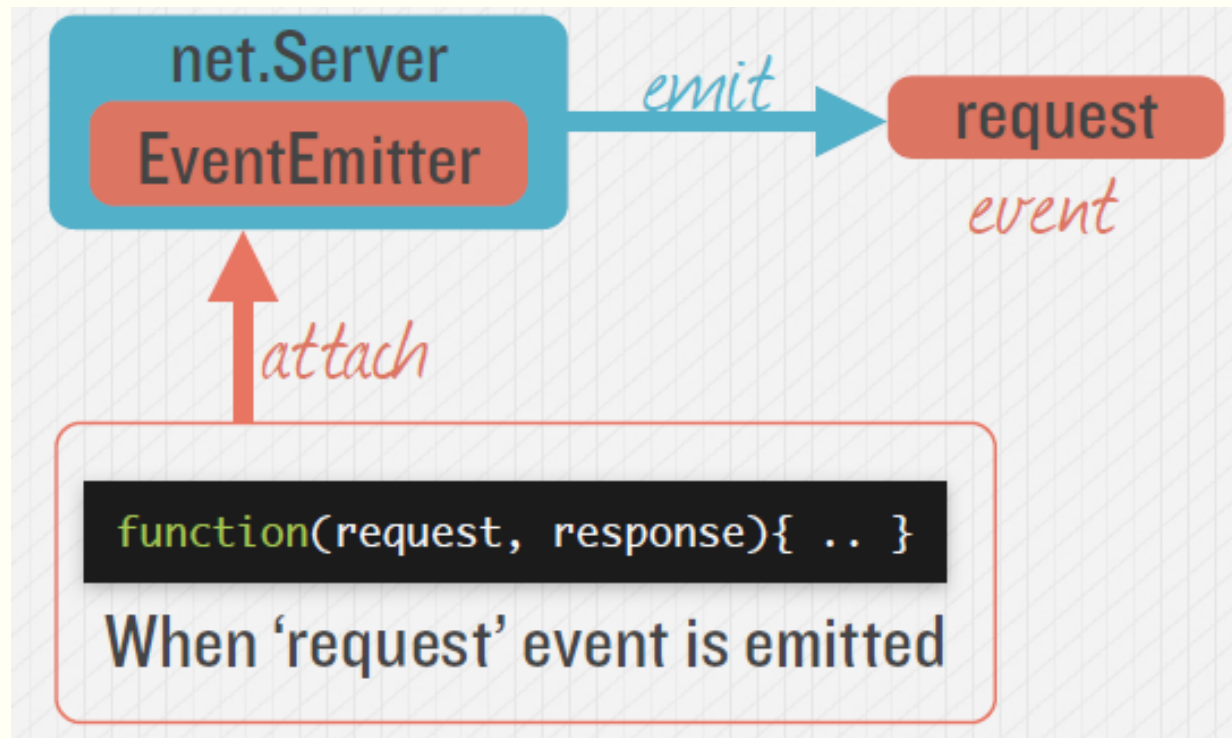
- Many objects in Node emit events



# EVENTS IN NODE

---

- Many objects in Node emit events



# CUSTOM EVENT EMITTERS

---

- **EventEmitter** class

- It allows you to listen for "events" and assign actions to run when those events occur.
- The principles EventEmitter is based on have been called the publish/subscribe model, because we can subscribe to events and then publish them.
- In Node, it's an alternative to deeply nested callbacks.
- A lot of Node methods are run asynchronously, which means that to run code after the method has finished, you need to pass a callback method to the function.
- Eventually, your code will look like a giant funnel. To prevent this, many node classes emit events that you can listen for.
- This allows you to organize your code the way you'd like to, and not use callbacks.

# EventEmitter

---

- we require the events module:

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

- This "on" method takes two parameters:
  - a) we start with the name of the event we're listening for:
  - b) The second parameter is the function that will be called when the event occurs.

The diagram illustrates the use of the EventEmitter module. It features two code snippets on a dark background. The first snippet shows the creation of a logger: `var logger = new EventEmitter();`. The second snippet shows the registration of an event listener: `logger.on('error', function(message){ console.log('ERR: ' + message); });`. To the right of the code, three red rounded rectangles are labeled 'error', 'warn', and 'info'. Above these rectangles, the word 'events' is written in red. A red arrow points from the 'error' rectangle to the 'error' string in the second code snippet, with the handwritten text 'listen for error event' next to it.

```
var logger = new EventEmitter();
```

*events*

error warn info

```
logger.on('error', function(message){  
  console.log('ERR: ' + message);  
});
```

*listen for error event*

# EventEmitter

---

- To fire the event, you pass the event name to the EventEmitter instance's **emit( )** method.
- You can pass data/info while firing events

```
logger.emit('error', 'Spilled Milk');
```

```
--> ERR: Spilled Milk
```

```
logger.emit('error', 'Eggs Cracked');
```

```
--> ERR: Eggs Cracked
```

# EventEmitter

---

- Other EventEmitter Methods

- `once`.

- It's just like the `on` method, except that it only works once. After being called for the first time, the listener is removed.
- `ee.once("firstConnection", function () {
  - console.log("You'll never see this again");});`
- `ee.emit("firstConnection");`
- `ee.emit("firstConnection");` // not picked up by any listeners



# STREAMS

---

- Streams can be readable, writeable, or both.
- Streams are **EventEmitters**, they emit several events at various points
- Example:
  - In a Node.js based HTTP server, request is a readable stream and response is a writable stream.
  - the fs module which lets you work with both readable and writable file streams

# Stream events

---

- Event: 'readable'
- When a chunk of data can be read from the stream, it will emit a 'readable' event

```
var readable = getReadableStreamSomehow();
readable.on('readable', function() {
  // there is some data to read now
});
```

# Stream events

---

- Event: 'data'
- Attaching a data event listener to a stream that has not been explicitly paused will switch the stream into flowing mode.
- Data will then be passed as soon as it is available.

```
var readable = getReadableStreamSomehow();
readable.on('data', function(chunk) {
  console.log('got %d bytes of data', chunk.length);
});
```

# Stream events

---

- Event: 'end'
  - This event fires when there will be no more data to read.
  - Note that the end event will not fire unless the data is completely consumed.
  - This can be done by switching into flowing mode, or by calling `read()` repeatedly until you get to the end.

```
readable.on('end', function() {  
  console.log('there will be no more data.');
```

```
});
```

# Stream events

---

- Event: 'close'
  - Emitted when the underlying resource has been closed. Not all streams will emit this.
- Event: 'error'
  - Emitted if there was an error receiving data.

# Stream events

---

- `readable.pause()`
  - This method will cause a stream in flowing mode to stop emitting data events, switching out of flowing mode.
  - Any data that becomes available will remain in the internal buffer.
- `readable.resume()`
  - This method will
  - cause the readable
  - stream to resume
  - emitting data events.

```
var readable = getReadableStreamSomehow();
readable.on('data', function(chunk) {
  console.log('got %d bytes of data', chunk.length);
  readable.pause();
  console.log('there will be no more data for 1 second');
  setTimeout(function() {
    console.log('now data will start flowing again');
    readable.resume();
  }, 1000);
});
```

# Readable Stream

---

- The function call `fs.createReadStream()` gives you a readable stream.
- Initially, the stream is in a static state. As soon as you listen to data event and attach a callback it starts flowing.
  - After that, chunks of data are read and passed to your callback.
  - When there is no more data to read (end is reached), the stream emits an end event

```
var fs = require('fs');
var readableStream = fs.createReadStream('readableExample1.js');
var content = '';

readableStream.on('data', function(chunk) {
  content += chunk;
});

readableStream.on('end', function() {
  console.log(content);
});
```

# Readable Stream

---

- The stream implementor decides how often data event is emitted.
- There is also another way to read from stream. You just need to call `read()` on the stream instance repeatedly until every chunk of data has been read.
- Note that the readable event is emitted when a chunk of data can be read from the stream.

```
var fs = require('fs');
var readableStream = fs.createReadStream('readableExample1.js');
var content = '';
var chunk;

readableStream.on('readable', function() {
  while ((chunk=readableStream.read()) != null) {
    content += chunk;
  }
});

readableStream.on('end', function() {
  console.log(content)
});
```



# Readable Stream

---

- Setting Encoding

- By default the data you read from a stream is a Buffer object. If you are reading strings this may not be suitable for you
- `readableStream.setEncoding('utf8');`
- As a result, the data is interpreted as utf8 and passed to your callback as string.

# Writable Streams

---

- Writable streams let you write data to a destination.
- Like readable streams, these are also EventEmitter and emit various events at various points.
- To write data to a writable stream you need to call write() on the stream instance. This function returns a Boolean value indicating if the operation was successful

```
var fs = require('fs');
var readableStream = fs.createReadStream('file1.txt');
var writableStream = fs.createWriteStream('file2.txt');

readableStream.setEncoding('utf8');

readableStream.on('data', function(chunk) {
  |   writableStream.write(chunk);
});
```

# Writable Streams

---

- When you don't have more data to write you can simply call `end()` to notify the stream that you have finished writing.
- Assuming `res` is an HTTP response object, you often do the following to send the response to browser:
  - `res.write('Some Data!!');`
  - `res.end('End.');`

# Piping

---

- Piping is a mechanism in which you can read data from the source and write to destination without managing the flow yourself.

```
var fs = require('fs');  
var readableStream = fs.createReadStream('file1.txt');  
var writableStream = fs.createWriteStream('file2.txt');  
readableStream.pipe(writableStream);
```

- `//pipe()` function to write the content of file1 to file2

# Chaining

---

- As pipe() manages the data flow for you, you should not worry about slow or fast data flow.
- This makes pipe() a neat tool to read and write data.
- You should also note that pipe() returns the destination stream. So, you can easily utilize this to chain multiple streams together.
- Assume that you have an archive and want to decompress it.

```
var fs = require('fs');  
var zlib = require('zlib');  
fs.createReadStream('input.txt.gz')           //read from the file input.txt.gz  
  .pipe(zlib.createGunzip())                  //un-gzip the content  
  .pipe(fs.createWriteStream('output.txt'));  
//write the un-gzipped content to the file
```

# Objectives

---

- Understand Handling HTTP requests with Node's API
- Understand RESTful web service
- Understand Serving static files
- Understand how to accept user input from forms
- Understand how to secure applications with HTTPS

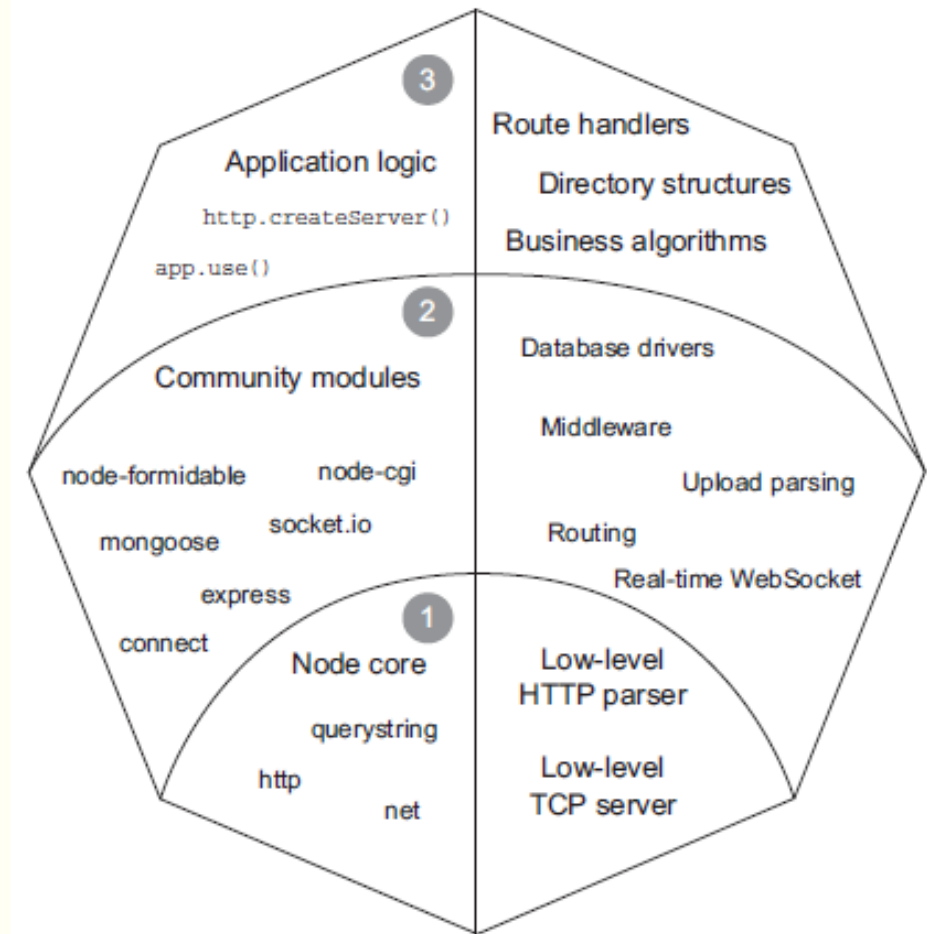
# Node HTTP server fundamentals

---

- Node's HTTP interface is low-level when compared with frameworks or languages
- such as PHP in order to keep it fast and flexible.
- High-level "sugar" APIs are left for third-party frameworks, such as Connect or Express, that greatly simplify the web application building process.

# Layers that make up a Node web application

- Node's core APIs are always lightweight and low-level.
- Community members take the low-level core APIs and create easy-to-use modules that allow you to get tasks done easily.
- The application logic layer is where your app is implemented.
  - The size of this layer depends on the number of community modules used and the complexity of the application.

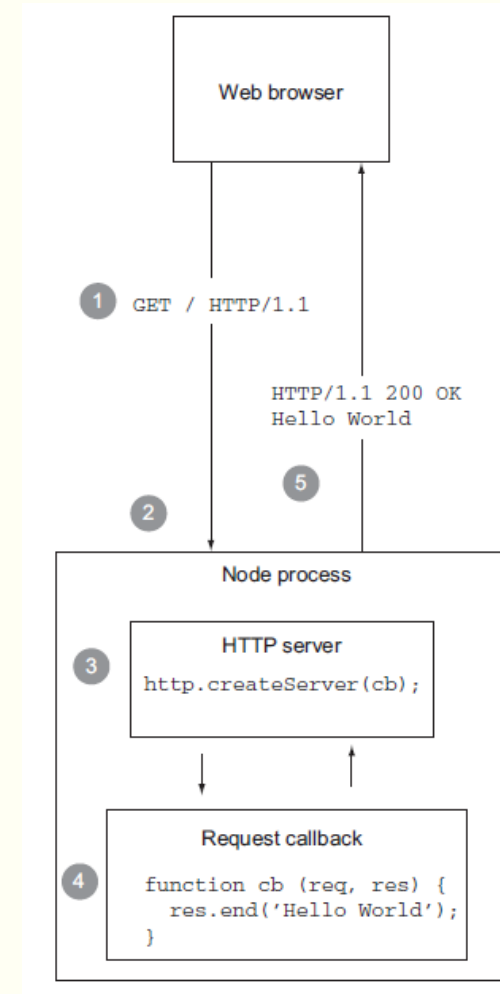




# Lifecycle of an HTTP request going through a Node HTTP server

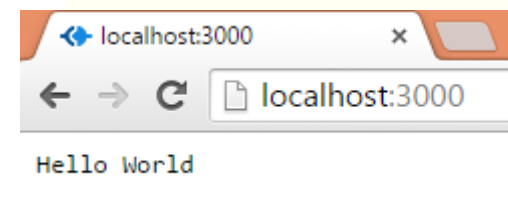
---

1. An HTTP client, like a web browser, initiates an HTTP request.
2. Node accepts the connection, and incoming request data is given to the HTTP server.
3. The HTTP server parses up to the end of the HTTP headers and then hands control over to the request callback.
4. The request callback performs application logic, in this case responding immediately with the text “Hello World.”
5. The request is sent back through the HTTP server, which formats a proper HTTP response for the client.



# Hello World HTTP server

```
/**
 * A basic HTTP server that responds with "Hello World"
 */
var http = require('http'); // http module
/*
 * To create an HTTP server, call the http.createServer() function
 * It accepts a single argument, a callback function,
 * that will be called on each HTTP request received by the server
 * This request callback receives, as arguments,
 * the request and response objects
 */
var server = http.createServer(function(req, res) {
  /*
   * call the res.write() method,
   * which writes response data to the socket
   */
  res.write('Hello World');
  /*
   * the res.end() method to end the response
   */
  res.end();
});
server.listen(3000); //listen for incoming requests.
```



# Response headers

---

- Setting response headers

```
var server = http.createServer(function(req, res) {  
  var body = 'Hello World';  
  res.setHeader('Content-Length', body.length);  
  res.setHeader('Content-Type', 'text/plain');  
  res.write(body);  
  res.end();  
});
```

## Setting the status code of an HTTP response

---

```
var server = http.createServer(function(req, res) {  
  var url = 'http://google.com';  
  var body = '<p>Redirecting to <a href="' + url + '">' +  
    url + '</a></p>';  
  res.setHeader('Location', url);  
  res.setHeader('Content-Length', body.length);  
  res.setHeader('Content-Type', 'text/html');  
  res.statusCode = 302;  
  res.end(body);  
});  
server.listen(3000); //listen for incoming requests.
```

# Request URL

---

- The requested URL can be accessed with the `req.url` property, which may contain several components depending on the request.
- To parse these sections, Node provides the `url` module, and specifically the **`.parse()` function**

```
G:\NodeEclipseMaterial_WS\streams>node
> require('url').parse('http://mindtree.com:3000/1?search=banuprakashc')
{ protocol: 'http:',
  slashes: true,
  auth: null,
  host: 'mindtree.com:3000',
  port: '3000',
  hostname: 'mindtree.com',
  hash: null,
  search: '?search=banuprakashc',
  query: 'search=banuprakashc',
  pathname: '/1',
  path: '/1?search=banuprakashc',
  href: 'http://mindtree.com:3000/1?search=banuprakashc' }
>
```

# Creating a static file server

---

```
var http = require('http');
var parse = require('url').parse;
var join = require('path').join;
var fs = require('fs');
/*
 * __dirname
 * The name of the directory that the
 * currently executing script resides in.
 */
var root = __dirname;
console.log(root);
var server = http.createServer(function(req, res) {
  var url = parse(req.url);
  var path = join(root, url.pathname);
  var stream = fs.createReadStream(path);
  stream.on('data', function(chunk) {
    res.write(chunk);
  });
  stream.on('end', function() {
    res.end();
  });
});
server.listen(3000);
```

- Each static file server has a root directory, which is the base directory files are served from.
- In the server you'll create, you'll define a root variable, which will act as the static file server's root directory

# Checking for a file's existence

---

```
fs.stat(path, function(err, stat) {  
  if (err) {  
    if ('ENOENT' == err.code) {  
      res.statusCode = 404;  
      res.end('Not Found');  
    } else {  
      res.statusCode = 500;  
      res.end('Internal Server Error');  
    }  
  } else {  
    res.setHeader('Content-Length', stat.size);  
    var stream = fs.createReadStream(path);  
    stream.pipe(res);  
    stream.on('error', function(err) {  
      res.statusCode = 500;  
      res.end('Internal Server Error');  
    });  
  }  
});
```

Check for  
file's existence

Some other  
error

Set Content-Length  
using stat object

# Restful services

---

- Representational State Transfer (REST) is a software architecture style consisting of guidelines and best practices for creating scalable web services.
- REST is a coordinated set of constraints applied to the design of components in a distributed hypermedia system that can lead to a more performant and maintainable architecture.
- REST has gained widespread acceptance across the Web as a simpler alternative to SOAP and WSDL-based Web services.
- RESTful systems typically, but not always, communicate over the Hypertext Transfer Protocol with the same HTTP verbs (GET, POST, PUT, DELETE, etc.) used by web browsers to retrieve web pages and send data to remote servers.
- The REST architectural style was developed by W3C Technical Architecture Group (TAG) in parallel with HTTP 1.1, based on the existing design of HTTP 1.0.
- The World Wide Web represents the largest implementation of a system conforming to the REST architectural style



# Restful services

---

```
/**
 * A basic RESTful service
 */
var http = require('http'); // http module
var url = require('url'); // url module
var items = [];
var server = http.createServer(function(req, res) {
  switch (req.method) {
    case 'POST':
      var item = '';
      req.setEncoding('utf8');
      req.on('data', function(chunk) {
        item += chunk;
      });
      req.on('end', function() {
        items.push(item);
        res.end('OK\n');
      });
      break;
```

# Restful services

---

```
case 'GET':  
    items.forEach(function(item, i) {  
        res.write(i + ') ' + item + '\n');  
    });  
    res.end();  
    break;
```

- To speed up responses, the Content-Length field should be sent with your response when possible.
- An optimized version of the GET handler could look something like this:  

```
var body = items.map(function(item, i){  
    return i + ') ' + item;  
}).join('\n');  
res.setHeader('Content-Length', Buffer.byteLength(body));  
res.setHeader('Content-Type', 'text/plain; charset="utf-8"');  
res.end(body);
```

# Restful services

---

- **Removing resources with DELETE requests**

- To accomplish this, the app will need to check the requested URL, which is how the HTTP client will specify which item to remove. In this case, the identifier will be the array index in the items array;
  - for example, DELETE /1 or DELETE /5

# Restful services

---

- Delete code

```
case 'DELETE':  
    var path = url.parse(req.url).pathname;  
    var i = parseInt(path.slice(1), 10);  
    if (isNaN(i)) {  
        res.statusCode = 400;  
        res.end('Invalid item id');  
    } else if (!items[i]) {  
        res.statusCode = 404;  
        res.end('Item not found');  
    } else {  
        items.splice(i, 1);  
        res.end('OK\n');  
    }  
    break;  
}
```

# Handling submitted form fields

---

- Typically two Content-Type values are associated with form submission requests:
  - application/x-www-form-urlencoded
    - The default for HTML forms
  - multipart/form-data
    - Used when the form contains files, or non-ASCII or binary data

# Handling submitted form fields

---

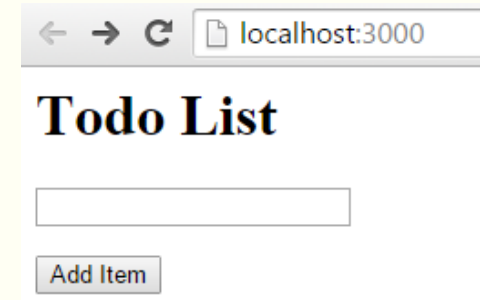
```
var http = require('http');
//Node's querystring module to parse the body
var qs = require('querystring');

var items = [];
var server = http.createServer(function(req, res) {
  if ('/' == req.url) {
    switch (req.method) {
      case 'GET':
        show(res);
        break;
      case 'POST':
        add(req, res);
        break;
    }
  } else {
    res.end('Not Found');
  }
});
server.listen(3000);
```

# Handling submitted form fields

---

- Handle get request
  - Display Form to accept user input



The screenshot shows a web browser window with the address bar displaying 'localhost:3000'. The page content includes a heading 'Todo List', a single-line text input field, and a button labeled 'Add Item'.

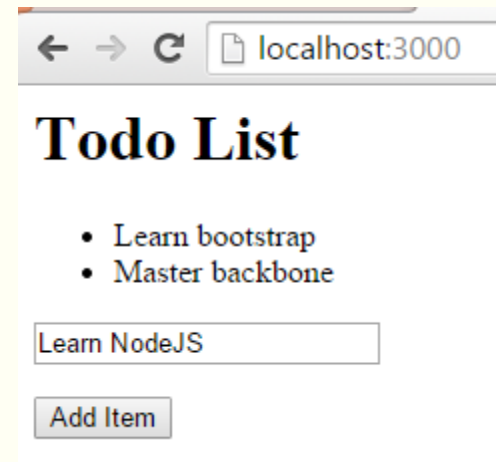
```
function show(res) {  
  var html = '<html><head><title>Todo List</title></head><body>'  
    + '<h1>Todo List</h1>'  
    + '<ul>' + items.map(function(item) {  
      return '<li>' + item + '</li>'  
    }).join('')  
    + '</ul>' + '<form method="post" action="/">'  
    + '<p><input type="text" name="item" /></p>'  
    + '<p><input type="submit" value="Add Item" /></p>'  
    + '</form></body></html>';  
  res.setHeader('Content-Type', 'text/html');  
  res.setHeader('Content-Length', Buffer.byteLength(html));  
  res.end(html);  
}
```

# Handling submitted form fields

---

- Handle form data
  - Use Node's **querystring** module to parse the body

```
function add(req, res) {  
  var body = '';  
  req.setEncoding('utf8');  
  req.on('data', function(chunk) {  
    body += chunk  
  });  
  req.on('end', function() {  
    var obj = qs.parse(body);  
    items.push(obj.item);  
    show(res);  
  });  
}
```





# HTTPS openssl

---

- To generate a private key, which we'll call key.pem,
- open up a command-line prompt and enter the following:
  - `openssl genrsa 1024 > key.pem`
- The private key is used to create the certificate. Enter the following to generate a certificate called key-cert.pem:
  - `openssl req -x509 -new -key key.pem > key-cert.pem`
- Unlike a private key, a certificate can be shared with the world; it contains a public key and information about the certificate holder.
- The public key is used to encrypt traffic sent from the client to the server

# HTTPS openssl

---

- Need to use HTTPS

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

var options = {
  key: fs.readFileSync('./key.pem'),
  cert: fs.readFileSync('./key-cert.pem')
};

https.createServer(options, function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}).listen(3000);
```

SSL key and cert  
given as options

options object is  
passed in first

https and http modules  
have almost identical APIs

# Objectives

---

- Understand how to bind node.js with mySql database
- Restful services using http and mysql module

# Relational database management systems

---

- Relational database management systems (RDBMSs) allow complex information to be stored and easily queried.
- RDBMSs have traditionally been used for relatively high-end applications, such as content management, customer relationship management, and shopping carts.
- Developers have many relational database options, but most choose open source databases, primarily because they're well supported, they work well, and they don't cost anything.
- MySQL and PostgreSQL have similar capabilities, and both are solid choices.
- MySQL is easier to set up and has a larger user base

# The node.js driver for mysql

---

- Install
  - `npm install mysql`
  - This is a node.js driver for mysql.
  - It is written in JavaScript, does not require compiling, and is 100% MIT licensed

# Connecting to MySQL

---

- Fill in the host, user, password, and database settings with those that correspond to your MySQL configuration

```
var mysql = require("mysql"); // mysql module
var connection = mysql.createConnection({
  host : 'localhost',
  user : 'root',
  password : 'Welcome123',
  database : 'northwind'
});
```

Require  
MySQL API

Connect to MySQL

# Few important Connection Options

---

When establishing a connection, you can set the following options:

- `host` : The hostname of the database you are connecting to. (Default: `localhost` )
- `port` : The port number to connect to. (Default: `3306`)
- `user` : The MySQL user to authenticate as.
- `password` : The password of that MySQL user.
- `database` : Name of the database to use for this connection
- `connectTimeout` : The milliseconds before a timeout occurs during the initial connection to the MySQL server. (Default: `10000` )
- `dateStrings` : Force date types (`TIMESTAMP`, `DATETIME`, `DATE`) to be returned as strings rather than inflated into JavaScript Date objects. (Default: `false` )
- `debug` : Prints protocol details to stdout. (Default: `false` )

# Performing queries

---

- The most basic way to perform a query is to call the `.query()` method on an object
- The simplest form of `.query()` is
  - `.query(sqlString, callback)` , where a SQL string is the first argument and the second is a callback:

```
connection.query('SELECT * FROM books WHERE author = "David",
```

```
function (error, results, fields) {
```

```
    // error will be an Error if one occurred during the query
```

```
    // results will contain the results of the query
```

```
    // fields will contain information about the returned results fields (if any)
```

```
});
```



# Performing queries

---

- Using IN parameters [ Placeholders]
  - `.query(sqlString, values, callback)`

```
connection.query('SELECT * FROM books WHERE author = ?',  
    ['David'],  
    function (error, results, fields) {  
  
    });
```

# Sample code to get json data from “suppliers” table

---

```
var http = require('http'); // http module
var mysql = require("mysql"); // mysql module
var server = http.createServer(function(req, res) {
  var connection = mysql.createConnection({
    host : 'localhost',
    user : 'root',
    password : 'Welcome123',
    database : 'northwind'
  });
  var query = "SELECT * FROM suppliers";
  query = mysql.format(query);
  connection.query(query, function(err, rows) {
    if (err) {
      res.write("Error executing MySQL query");
    } else {
      res.write(JSON.stringify(rows));
    }
  });
  connection.end();
});
server.listen(3000); // listen for incoming requests.
```

- Every method you invoke on a connection is queued and executed in sequence.
- Closing the connection is done using `end()` which makes sure all remaining queries are executed before sending a quit packet to the mysql server.

# Sample code to insert form-data into table

---

- Helper function to parse HTTP POST data
  - Converts form parameters into JSON data

```
var qs = require('querysting'); // querysting
parseReceivedData = function(req, cb) {
  var body = '';
  req.setEncoding('utf8');
  req.on('data', function(chunk) {
    body += chunk
  });
  req.on('end', function() {
    var data = qs.parse(body);
    cb(data);
  });
};
```

# Sample code to insert form-data into table

---

- Insert Data to EMP table

```
var insertQuery = "INSERT into EMP" +  
    " (EMPNO,ENAME,JOB,HIREDATE) " +  
    " values(?,?,?,?)";  
parseReceivedData(req, function(emp) {  
    connection.query(insertQuery, [emp.EMPNO, emp.ENAME,  
        emp.JOB,emp.HIREDATE], function(err) {  
        if (err)  
            throw err;  
        connection.on('end', function() {  
            res.end();  
        });  
        connection.end();  
    });  
});
```

# Sample code to insert json-data into table

---

- {"EMPNO":63,"ENAME":"T","HIREDATE":"2004-4-4"}

```
var insertQuery = "INSERT into EMP" + " (EMPNO,ENAME,JOB,HIREDATE) "
    + " values(?,?,?,?)";
var emp = '';
req.on('data', function(chunk) {
    emp += chunk;
});
req.on('end', function() {
    console.log(emp);
    emp = JSON.parse(emp);
    connection.query(insertQuery, [ emp.EMPNO, emp.ENAME, emp.JOB,
        emp.HIREDATE ], function(err) {
        if (err)
            throw err;
        connection.on('end', function() {
            res.end();
        });
        connection.end();
    });
});
```

# Escaping query values

---

- In order to avoid SQL Injection attacks, you should always escape any user provided data before using it inside a SQL query.
- You can do so using the `mysql.escape()` , `connection.escape()` or `pool.escape()` methods
  - ```
var sql = 'SELECT * FROM users WHERE id = ' +  
    connection.escape(userId);
```
- Alternatively, you can use `?` characters as placeholders for values you would like to have escaped like this:

```
connection.query('SELECT * FROM users WHERE id = ?', [userId], function(err, results) {  
    // ...  
});
```

# Escaping query values

---

- Different value types are escaped differently, here is how:
  - Numbers are left untouched
  - Booleans are converted to `true` / `false`
  - Date objects are converted to `'YYYY-mm-dd HH:ii:ss'` strings
  - Buffers are converted to hex strings, e.g. `X'0fa5'`
  - Strings are safely escaped
  - Arrays are turned into list, e.g. `['a', 'b']` turns into `'a', 'b'`
  - Nested arrays are turned into grouped lists (for bulk inserts), e.g. `[['a', 'b'], ['c', 'd']]` turns into `('a', 'b'), ('c', 'd')`
  - Objects are turned into `key = 'val'` pairs for each enumerable property on the object. If the property's value is a function, it is skipped; if the property's value is an object, `toString()` is called on it and the returned value is used.
  - `undefined` / `null` are converted to `NULL`
  - `NaN` / `Infinity` are left as-is. MySQL does not support these, and trying to insert them as values will trigger MySQL errors until they implement support.

# Transactions

---

- beginTransaction(), commit() and rollback() are simply convenience functions that execute the START TRANSACTION, COMMIT, and ROLLBACK commands respectively

```
connection.beginTransaction(function(err) {
  if (err) { throw err; }
  connection.query('SQL...',
    function(err, result) {
      if (err) {
        connection.rollback(function() {
          throw err;
        });
      }
    }
  );
  connection.query('SQL..',
    function(err, result) {
      if (err) {
        connection.rollback(function() {
          throw err;
        });
      }
    }
  );
});
```

```
connection.commit(function(err) {
  if (err) {
    connection.rollback(function() {
      throw err;
    });
  }
  console.log('success!');
});
});
});
});
```



# Objectives

---

- Understand NodeJS express-4 module
- Build RESTful application using node express

# Express

---

- What's Express?
  - NodeJS based web framework
  - Asynchronous

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

app.get('/', function(req,res) {
  res.send('Hello World');
});

app.listen(3000);
```

# Express

---

- Installation
  - Install node
  - Install npm
  - npm install express

# Configuration for Development and Production env

---

```
// development error handler
// will print stacktrace
if (app.get('env') === 'development') {
  app.use(function(err, req, res, next) {
    res.status(err.status || 500);
    res.render('error', {
      message: err.message,
      error: err
    });
  });
}

// production error handler
// no stacktraces leaked to user
app.use(function(err, req, res, next) {
  res.status(err.status || 500);
  res.render('error', {
    message: err.message,
    error: {}
  });
});
```

# Setup Express project

---

- `npm install serve-favicon morgan method-override express-session body-parser multer errorhandler express@latest jade@latest --save`

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "body-parser": "^1.5.2",
    "errorhandler": "^1.1.1",
    "express": "^4.8.0",
    "express-session": "^1.7.2",
    "jade": "^1.5.0",
    "method-override": "^2.1.2",
    "morgan": "^1.2.2",
    "multer": "^0.1.3",
    "serve-favicon": "^2.0.1"
  }
}
```

# Hello world example

---

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

app.get('/', function (req, res) {
  res.send('Hello World!');
});

var server = app.listen(3000, function () {
  var host = server.address().address;
  var port = server.address().port;
  console.log('Listening at http://%s:%s', host, port);
});
```

# Routing

---

- Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).
- Each route can have one or more handler functions, which is / are executed when the route is matched.
- Route definition takes the following structure `app.METHOD (PATH, HANDLER)`, where `app` is an instance of `express`, `METHOD` is an HTTP request method, `PATH` is a path on the server, and `HANDLER` is the function executed when the route is matched.

# Routing example

---

```
// respond with "Hello World!" on the homepage
app.get('/', function (req, res) {
  res.send('Hello World!');
});

// accept POST request on the homepage
app.post('/', function (req, res) {
  res.send('Got a POST request');
});

// accept PUT request at /user
app.put('/user', function (req, res) {
  res.send('Got a PUT request at /user');
});

// accept DELETE request at /user
app.delete('/user', function (req, res) {
  res.send('Got a DELETE request at /user');
});
```



# Route paths

---

```
// will match request to the root
app.get('/', function (req, res) {
  res.send('root');
});

// will match requests to /about
app.get('/about', function (req, res) {
  res.send('about');
});

// will match request to /random.text
app.get('/random.text', function (req, res) {
  res.send('random.text');
});
```

# Route paths

---

```
// will match acd and abcd
app.get('/ab?cd', function(req, res) {
  res.send('ab?cd');
});

// will match abcd, abbcd, abbbcd, and so on
app.get('/ab+cd', function(req, res) {
  res.send('ab+cd');
});

// will match abcd, abxcd, abRABDOMcd, ab123cd, and so on
app.get('/ab*cd', function(req, res) {
  res.send('ab*cd');
});

// will match /abe and /abcde
app.get('/ab(cd)?e', function(req, res) {
  res.send('ab(cd)?e');
});
```

# Create modular routes

---

- Chained route handlers for a route path can be created using `app.route()`. Since the path is specified at a single location, it helps to create modular routes and reduce redundancy and typos

```
app.route('/book')
  .get(function(req, res) {
    res.send('Get a random book');
  })
  .post(function(req, res) {
    res.send('Add a book');
  })
  .put(function(req, res) {
    res.send('Update the book');
  });
```

# express.Router

---

- The `express.Router` class can be used to create modular mountable route handlers.
- A Router instance is a complete middleware and routing system; for this reason it is often referred to as a “mini-app”.

```
//birds.js
// middleware specific to this router
router.use(function timeLog(req, res, next) {
  console.log('Time: ', Date.now());
  next();
});
// define the home page route
router.get('/', function(req, res) {
  res.send('Birds home page');
});
// define the about route
router.get('/about', function(req, res) {
  res.send('About birds');
});

module.exports = router;
```

```
var birds = require('./birds');
app.use('/birds', birds);
```

# Serving static files in Express

---

- Serving files, such as images, CSS, JavaScript and other static files is accomplished with the help of a built-in middleware in
  - Express - `express.static`.
- Pass the name of the directory, which is to be marked as the location of static assets, to the `express.static` middleware to start serving the files directly.
- For example, if you keep your images, CSS, and JavaScript files in a directory named `public`, you can do this:
  - `app.use(express.static('public'));`

# Serving static files in Express

---

- If you want to use multiple directories as static assets directories, you can call the `express.static` middleware multiple times:
  - `app.use(express.static('public'));`
  - `app.use(express.static('files'));`
- The files will be looked up in the order the static directories were set using the `express.static` middleware.

# Objectives

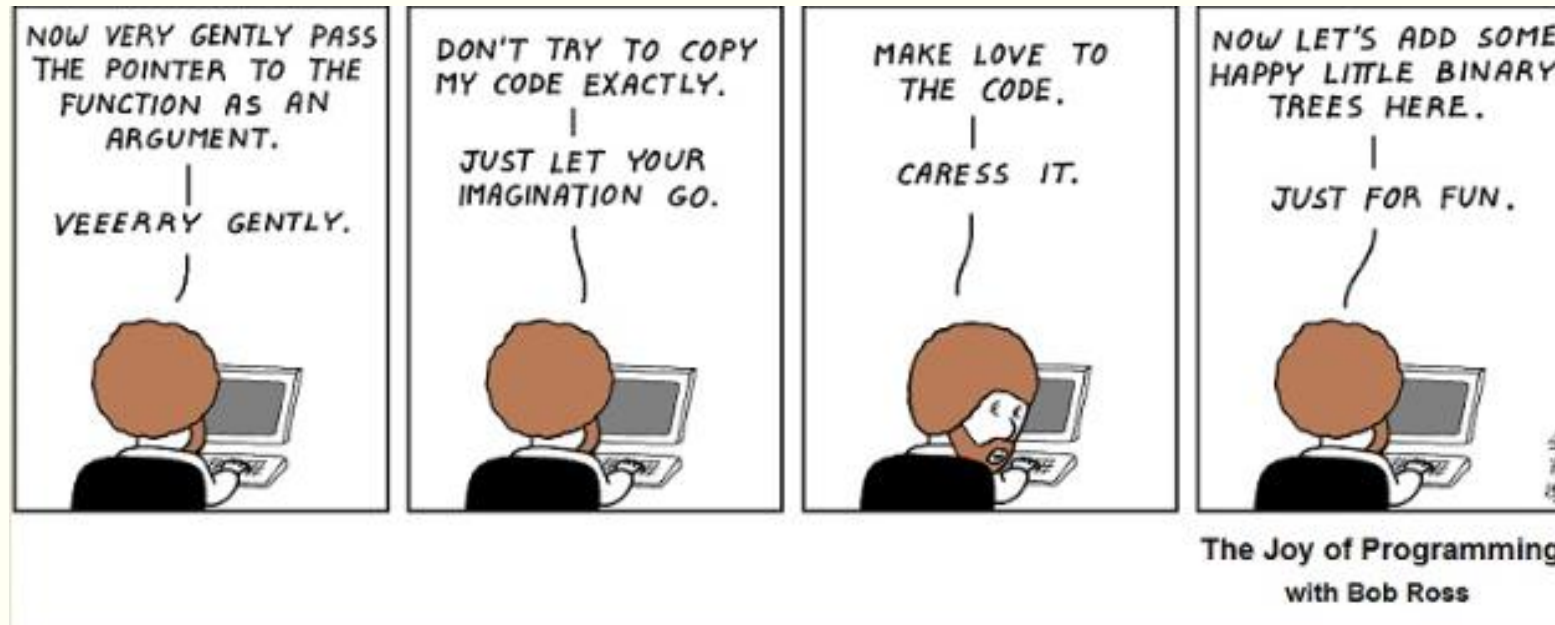
---

- Understand TDD and BDD
- Understand different Testing frameworks used to test NodeJS modules

# Why Write Tests?

---

- The Joy of Programming





# Why Write Tests?

---

- Programming should not be frustrating



## Why Write Tests?

---

The biggest Joy in programming is knowing  
“**What you write**” does “**What you intended**”  
to do, and it **doesn't break** anything else.

# Why Write Tests?

---

- Most common excuses to not write tests
  - Laziness: “Writing tests is such a unpleasant task!”
  - Perceived busyness: “We have too many things to do. We have targets to hit!!!”
  - Overconfidence: “This function is really easy. I can handle this.”
  - Ego: “It’ll definitely work. I’m a Programmer”.

# Why Write Tests?

---

- What are the consequences?



# Why Write Tests?

---

- What Programmers deal with?
  - Old code breaks when you write a new code
  - Your code just broke other people's code
  - Code that you wrote is slow
  - Spend hours debugging your code and other people's code
  - What you've done is not what is supposed to do.
  - and ....

# Why Write Tests?

---

- Whose fault is it?
  - Yours, of course



# Why Write Tests?

---

- If you don't want to deal with all that, write tests.



# TDD (Test Driven Development)

---

- Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle:
  - first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards.
  - Kent Beck, who is credited with having developed or 'rediscovered' the technique, stated in 2003 that TDD encourages simple designs and inspires confidence



# TDD (Test Driven Development)

---

- Step 1: Write Test
- Step 2: Run the test ( It will fail, It's Okay)
- Step 3: Write the code to make the test pass
- Step 4: Celebrate that your code works wonderfully.

# BDD (Behaviour Driven Development)

---

- It includes the practice of writing tests first, but focuses on tests which describe behaviour, rather than tests which test a unit of implementation
  - Step 1: Gather requirements from the business side
  - Step 2: Write the test cases to meet the pre-defined business requirements
  - Step 3: Write the code to make the test pass
  - Step 4: Show it to your business partner

# How to test JavaScript?

---



# Mocha

---

- Mocha is a feature-rich JavaScript test framework running on node.js and the browser, making asynchronous testing simple and fun.
- Mocha tests run serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases.
  - Runs on nodeJS/ Browser
  - Supports TDD and BDD
  - Choose any assertion library
  - Choose any Mocking library
  - Async and Promise support
- Installation
  - `npm install -g mocha`

# The Basics

---

- The package.json
- devDependencies
  - If someone is planning on downloading and using your module in their program, then they probably don't want or need to download and build the external test or documentation framework that you use.
  - In this case, it's best to map these additional items in a devDependencies object.

```
  "devDependencies": {  
    "mocha": "^2.2.5"  
  },  
  "scripts": {  
    "start": "node ./bin/www",  
    "test": "mocha"  
  },  
}
```

# The Basics

---

- Then run `npm install` and you're almost ready to use Mocha.
- The Mocha script will be in `./node_modules/.bin/mocha`
- Create "test" folder. == > folder which contains all tests
- The "BDD" interface provides `describe()`, `it()`, `before()`, `after()`, `beforeEach()`, and `afterEach()`.
- The "TDD" interface provides `suite()`, `test()`, `setup()`, and `teardown()`.

# Example BDD

---

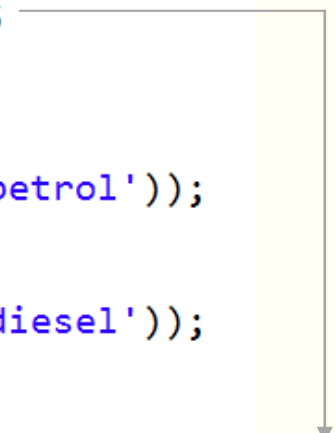
- test/arrayTest.js

```
var assert = require("assert")
describe('Array', function(){
  describe('#indexOf()', function(){
    it('should return -1 when the value is not present', function(){
      assert.equal(-1, [1,2,3].indexOf(5));
      assert.equal(-1, [1,2,3].indexOf(0));
    })
  })
})
```

# Testing Synchronous Code

---

```
var assert = require("assert")
var Vehicle = require("../public/js/Vehicle").Vehicle;
describe('Vehicle', function() {
  describe('Test Fuel Cost', function() {
    it('should be 75.00', function() {
      assert.equal(75.00, Vehicle.getFuelCost('petrol'));
    });
    it('should be 54.50', function() {
      assert.equal(54.50, Vehicle.getFuelCost('diesel'));
    });
  });
})
```



```
exports.Vehicle = (function(){
  function getFuelCost(type) {
    if(type === 'petrol') {
      return 75.00;
    } else if(type === 'diesel') {
      return 54.50;
    }
  }
  return {
    getFuelCost : getFuelCost
  }
})();
```



# Asynchronous code

---

- Simply invoke the callback when your test is complete.
- By adding a callback (usually named done) to it() Mocha will know that it should wait for completion.

```
describe('User', function() {  
  describe('#save()', function() {  
    it('should save without error', function(done) {  
      var user = new User('Luna');  
      user.save(function(err) {  
        if (err) throw err;  
        done();  
      });  
    });  
  });  
})  
})
```

# Hooks

---

- Mocha provides the hooks `before()`, `after()`, `beforeEach()`, `afterEach()`, that can be used to set up preconditions and clean up your tests

```
describe('hooks', function() {  
  before(function() {  
    // runs before all tests in this block  
  })  
  after(function(){  
    // runs after all tests in this block  
  })  
  beforeEach(function(){  
    // runs before each test in this block  
  })  
  afterEach(function(){  
    // runs after each test in this block  
  })  
  // test cases  
})
```

# Should assertion library

---

|                                                     |                                              |
|-----------------------------------------------------|----------------------------------------------|
| <code>x.should.be.ok</code>                         | <code>// truthiness</code>                   |
| <code>x.should.be.true</code>                       | <code>// === true</code>                     |
| <code>x.should.be.false</code>                      | <code>// === false</code>                    |
| <code>x.should.be.empty</code>                      | <code>// length == 0</code>                  |
| <code>x.should.be.within(y,z)</code>                | <code>// range</code>                        |
| <code>x.should.be.a(y)</code>                       | <code>// typeof</code>                       |
| <code>x.should.be[.an].instanceOf(y)</code>         | <code>// instanceof</code>                   |
| <code>x.should.be.above(n)</code>                   | <code>// &gt; val</code>                     |
| <code>x.should.be.below(n)</code>                   | <code>// &lt; val</code>                     |
| <code>x.should.eql(y)</code>                        | <code>// ==</code>                           |
| <code>x.should.equal(y)</code>                      | <code>// ===</code>                          |
| <code>x.should.match(/y/)</code>                    | <code>// regexp match</code>                 |
| <code>x.should.have.length(y)</code>                | <code>// .length == y</code>                 |
| <code>x.should.have.property(prop[, val])</code>    | <code>// prop exists</code>                  |
| <code>x.should.have.ownProperty(prop[, val])</code> | <code>// prop exists (immediate)</code>      |
| <code>x.should.have.status(code)</code>             | <code>// .statusCode == y</code>             |
| <code>x.should.have.header(field[, val])</code>     | <code>// .header with field &amp; val</code> |
| <code>x.should.include(y)</code>                    | <code>// x.indexOf(y) != -1</code>           |
| <code>x.should.throw([string   /regexp/])</code>    | <code>// thrown exception</code>             |

## *Negation:*

- `x.should.not.be.ok`

## *Chaining:*

- `x.should.be.a('string').and.have.length(5)`

## *Implementation:*

- **should** added to Object as property
- Therefore **x** must not be null or undefined
- Use **should.exist(x)** to test first where needed.

# Testing Express Code

---

- supertest is a Super-agent driven library for testing.
- supertest works by starting an express application and running test against it.
- This type of testing is often referred to as end-to-end testing or some times integration testing.
- This style of testing has the benefit of testing your public api where unit test test's individual internal units.

package.json

```
"devDependencies": {  
  "mocha": "*",  
  "should": ">= 0.0.1",  
  "supertest": "0.3.x"  
}
```

# Testing Express Code

---

- supertest takes the express app that it will be testing as an argument, therefore we must export our app from app.js

**app.js**

```
//...
```

```
module.exports = app;
```

**test file:**

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

```
var request = require('supertest');
```

# Testing Express Code

---

```
var assert = require("assert")
var employees = require('../routes/employees');
var app = require('../app');
var request = require('supertest');
var should = require('should');

describe('Employees CRUD', function(done) {
  describe('Get Employees', function(done) {
    var records;
    before(function(done){
      request(app).get('/employees/').end(function(err,res){
        should.not.exist(err);
        records = res.body;
        done();
      });
    })
    it('should return 18 records', function(done) {
      assert.equal(18, records.length);
      done();
    });
    after(function(){});
  })
})
```

# Testing Express Code

---

- POST Restful service

```
/* ADD employee */
router.post('/', function(req, res, next) {
  connection.query("insert into EMP(EMPNO,ENAME,HIREDATE) values(?,?,?)"
    ,[req.body.EMPNO,req.body.ENAME,req.body.HIREDATE],
    function(err, fields) {
      if(!err){
        res.json({message:"Employee inserted successfully!!!"});
      } else {
        console.log(err);
        res.json({message:"Unable to insert employee"});
      }
    });
});
```

# Testing Express Code

---

```
describe('POST Employee', function(done) {
  var response;
  before(function(done){
    var employee = {"EMPNO":9000,"ENAME":"TEST","HIREDATE":"2003-3-3"};
    request(app).post('/employees/')
      .send(employee)
      .expect(200)
      .expect('Content-Type', /json/)
      .end(function(err,res){
        should.not.exist(err);
        response = res.body;
        done();
      });
  })
  it('should insert employee 9000', function(done) {
    assert.equal(response.message, "Employee inserted successfully!!!");
    done();
  });
  after(function(){});
})
```



# References

---

- <https://github.com/mochajs/mocha/>

# Objectives

---

- Understand Session Handling using NodeJS
- Understand Using Passport for Authentication
- Understand OAuth

# HTTP Protocol

---

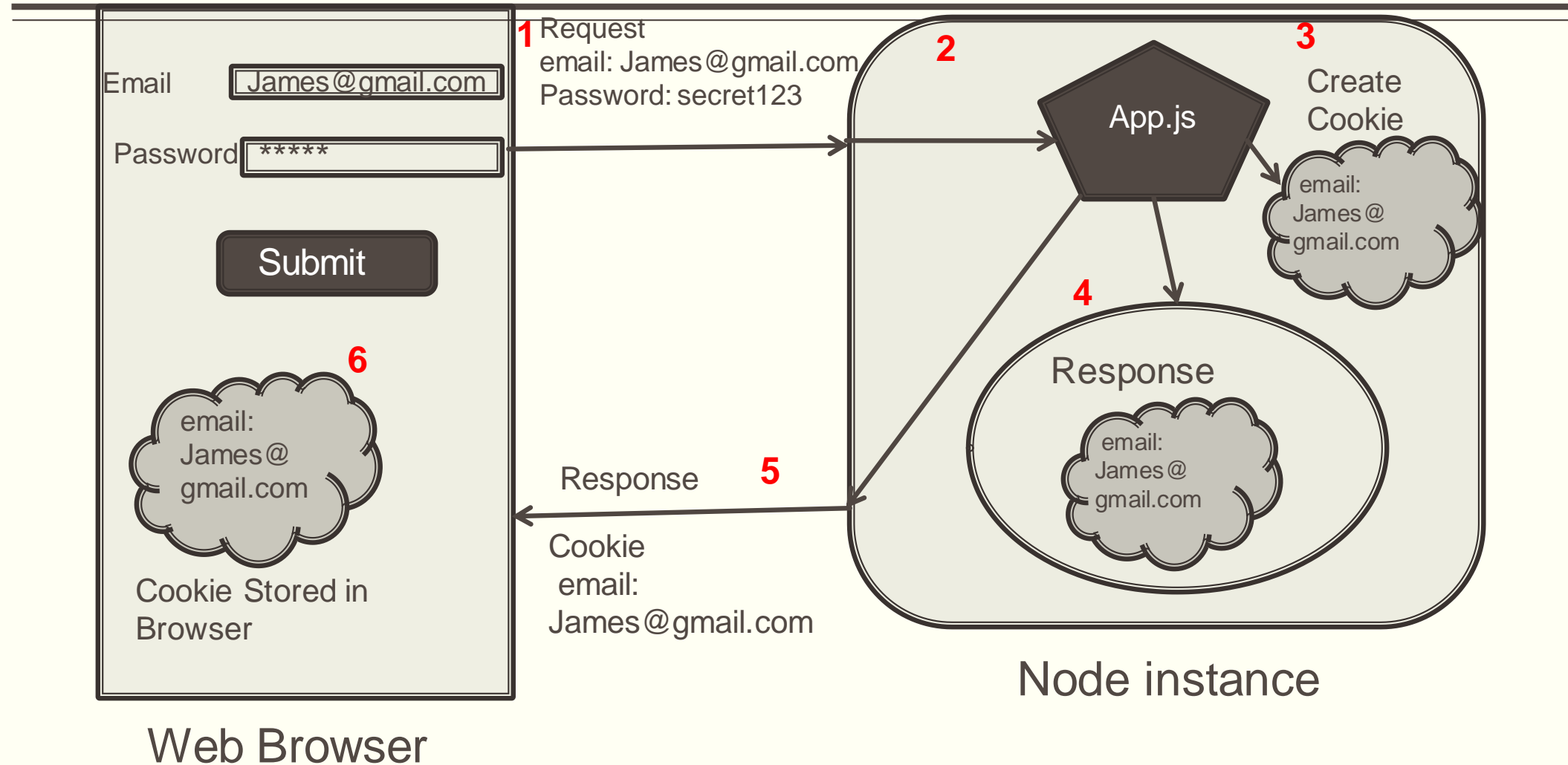
- HTTP Protocol is a stateless protocol which means that each page request is considered independent of any other request.
  - A web server does not understand if a page request comes from someone who has already requested a page or if the person is visiting the page for the first time.
  - It treats each request in the same way.
- In some scenarios, e.g. ecommerce and banking web applications this is inconvenient and we need some way to tie every page request together as a session of a single visitor.

# Session Tracking

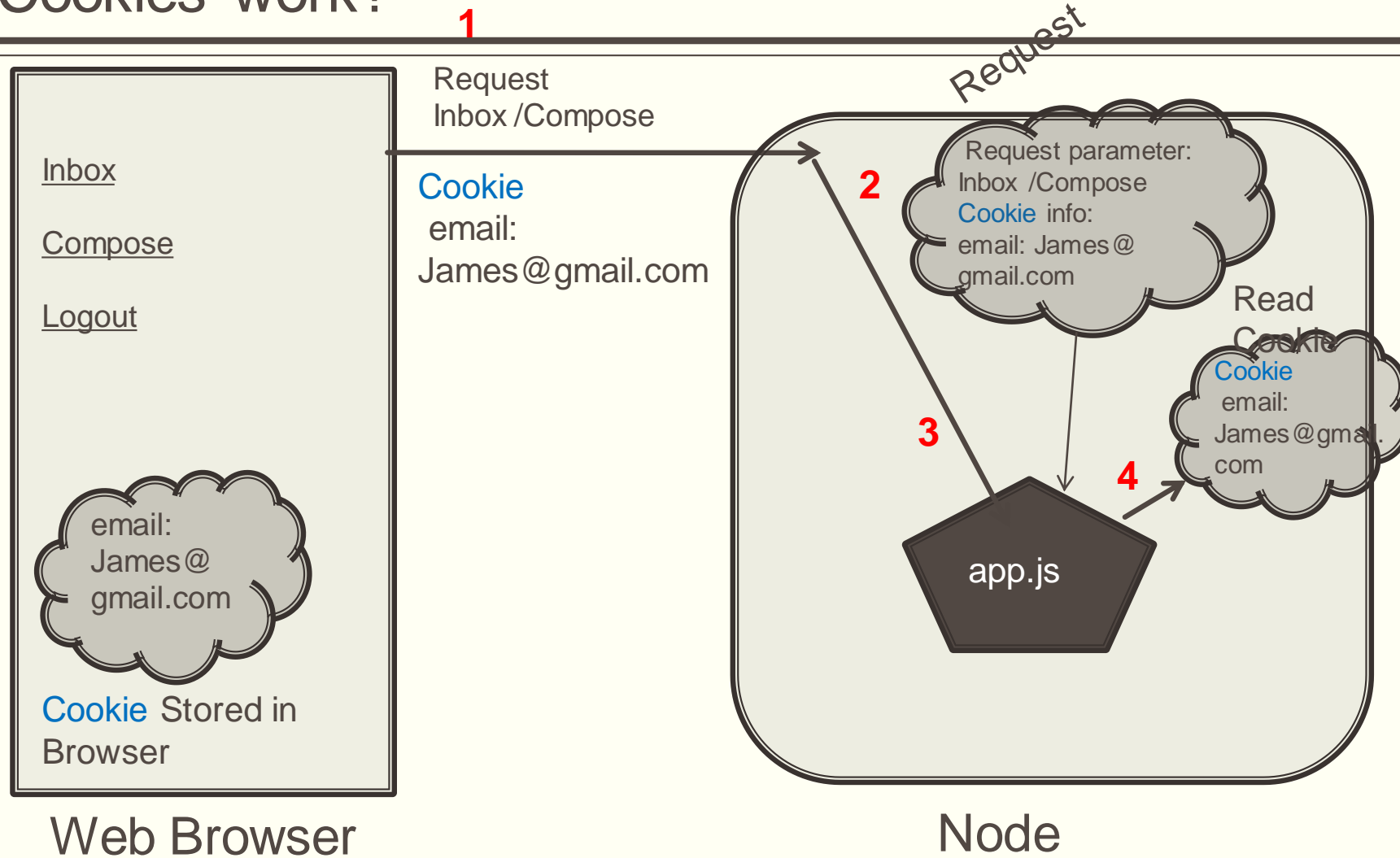
---

- Session tracking is a mechanism which helps the servers to maintain the conversational state of a client.
  - **Cookies** can be used for session tracking.
  - A **cookie** is a bit of information sent by a web server to a browser that can later be read back from that browser.
  - Browser receives a cookie and it saves the cookie.
  - Browser sends the cookie back to the server each time it accesses a page on that server

# How Cookie Works?



# How Cookies work?



# Express Session

---

- Include “cookie-parser” and “express-session” modules
- Set secret for cookie-handling

```
var cookieParser = require('cookie-parser');
var session = require('express-session');
app.use(cookieParser());

app.use(session({secret: "abracadabra", proxy: true,
  resave: true,
  saveUninitialized: true
}));
```

# Express Session

---

- Set Filters before any Routers

```
// Filter for only allowing authenticated requests through
app.all('*', function(req, res, next) {
  console.log(req.originalUrl);
  if (req.originalUrl.indexOf("scripts") !== -1
      || req.originalUrl === "/login.html"
      || req.originalUrl === "/users/auth"
      || (req.session.user !== null
          && req.session.user !== undefined)) {
    console.log("Proceeding next...");
    next();
  } else {
    console.log("Error in auth...");
    res.redirect("/login.html");
  }
});

app.use(express.static(path.join(__dirname, 'public')));
app.use('/', routes);
app.use('/users', users);
```



# Express-Session [ Handling Login and Logout]

---

```
<form name="Login" action="/users/auth" method="post">
  Email: <input type="text" name="email"/>
  Password: <input type="password" name="password"/>
  <input type="submit" value="Login"/>
</form>
```

```
/* Authenticate user. */
router.post('/auth', function(req, res, next) {
  console.log("/auth === > " + req.body.email);
  var email = req.body.email;
  var pwd = req.body.password;
  var ses=req.session;
  if( email === 'banu_prakash@mindtree.com'
      && pwd === 'test') {
    ses.user = email;
    res.redirect('/index.html');
  } else {
    ses.user=null;
    res.redirect('/login.html');
  }
});
```

# Express-Session [ Handling Login and Logout]

---

- Logout

```
router.get('/logout', function(req, res, next) {  
  console.log("/Logout === > " + req.body.email);  
  var ses=req.session;  
  if( ses !== null && ses !== undefined) {  
    ses.user = null;  
    ses.destroy();  
  }  
  res.redirect('/login.html');  
});
```

- Refer: <https://github.com/expressjs/session>

# Passport Authentication

---

- **Implementing Facebook Authentication**

- npm install passport –g
- npm install -g passport-local
- npm install passport-facebook

# Passport

---

- Passport is authentication middleware for Node.js.
- Extremely flexible and modular, Passport can be unobtrusively dropped in to any Express-based web application.
- A comprehensive set of strategies support authentication using a username and password, Facebook, Twitter, and more.

# Passport: Strategies

---

- Passport recognizes that each application has unique authentication requirements. Authentication mechanisms, known as *strategies*, are packaged as individual modules.
- Applications can choose which strategies to employ, without creating unnecessary dependencies

```
var passport = require('passport');
var LocalStrategy = require('passport-local').Strategy;

app.use(passport.initialize());
app.use(passport.session());

passport.use(new LocalStrategy(function(username, password, done) {
    if ("test123" !== password) {
        return done(null, false);
    }
    return done(null, username);
})));
```

# Passport: Redirects

---

- Upon successful authentication, the user will be redirected to the home page.
- If authentication fails, the user will be redirected back to the login page for another attempt.
- In this case, the redirect options override the default behaviour.

```
app.post('/login', passport.authenticate('local', {
  successRedirect : '/loginSuccess',
  failureRedirect : '/loginFailure'
}));

app.get('/loginFailure', function(req, res, next) {
  res.send('Failed to authenticate');
});

app.get('/loginSuccess', function(req, res, next) {
  res.send('Successfully authenticated');
});
```

# Passport :Flash Messages

---

- Redirects are often combined with flash messages in order to display status information to the user.

```
app.post('/login',  
  passport.authenticate('local', { successRedirect: '/',  
                                   failureRedirect: '/login',  
                                   failureFlash: true })  
);
```

- Setting the failureFlash option to true instructs Passport to flash an error message using the message given by the strategy's verify callback, if any.
- This is often the best approach, because the verify callback can make the most accurate determination of why authentication failed.
- the flash message can be set specifically.
- `passport.authenticate('local', { failureFlash: 'Invalid username or password.' });`

# Passport : Sessions

---

- In a typical web application, the credentials used to authenticate a user will only be transmitted during the login request.
- If authentication succeeds, a session will be established and maintained via a cookie set in the user's browser.
- Each subsequent request will not contain credentials, but rather the unique cookie that identifies the session.
- In order to support login sessions, Passport will serialize and deserialize user instances to and from the session.



# Passport : Sessions

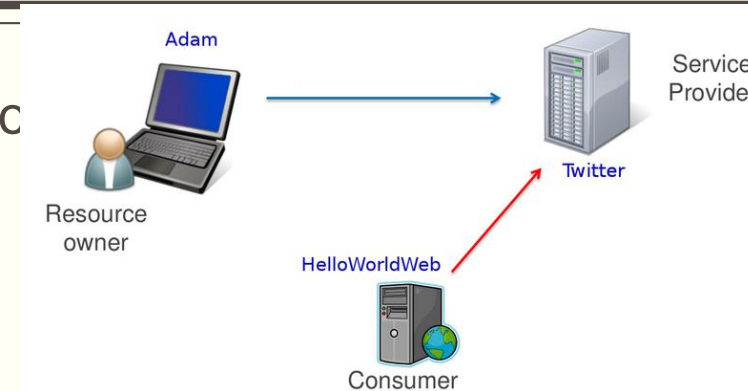
---

```
passport.serializeUser(function(user, done) {  
  done(null, user.id);  
});  
passport.deserializeUser(function(id, done) {  
  User.findById(id, function(err, user) {  
    done(err, user);  
  });  
});
```

- In this example, only the user ID is serialized to the session, keeping the amount of data stored within the session small.
- When subsequent requests are received, this ID is used to find the user, which will be restored to req.user.

# OAuth

- OAuth is a specification that defines secure authentication model on behalf of another user.
- The first party represents a user, in our case Adam, who is called in the OAuth terminology a Resource Owner. Adam has an account on Twitter.



Twitter represents the second party. This party is called a Service Provider.

Twitter offers a web interface that Adam uses to create new tweets, read tweets of others etc. Now, Adam uses our new web site, HelloWorldWeb, that displays the last tweet of the logged in user.

To do so, web site needs to have access to the Twitter account of Adam.

HelloWorldWeb site is a 3rd party application that wants to connect to Twitter and get Adam's tweets.

In OAuth, such party is called Consumer.

# OAuth

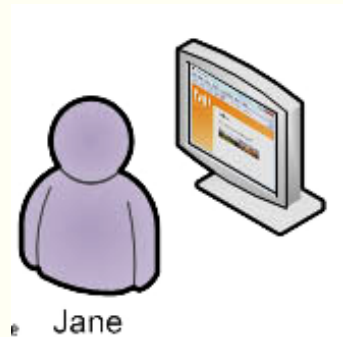
---

- Two versions of OAuth exists at the moment
  - *OAuth 1* defined by [OAuth 1.0 specification](#)
  - *OAuth 2* defined by [OAuth 2.0 specification](#).

# OAuth example

---

- *Jane wants to share some of her vacation photos with her friends.*
- *Jane uses Faji, a photo sharing site, for sharing journey photos.*
- *She signs into her faji.com account, and uploads two photos which she marks private*
- In OAuth terminology, Jane is the resource owner and Faji the server. The 2 photos Jane uploaded are the protected resources



# OAuth example

---

- *Jane visits beppa.com and begins to order prints. Beppa supports importing images from many photo sharing sites, including Faji. Jane selects the photos source and clicks Continue.*

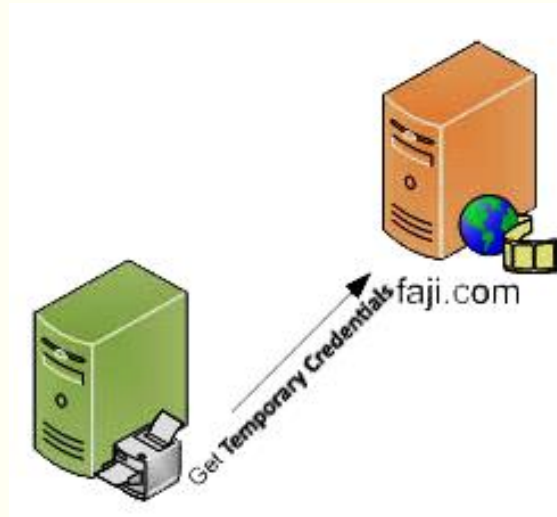


- In OAuth terminology, Beppa is the client. When Beppa added support for Faji photo import, a Beppa developer known in OAuth as a client developer obtained a set of client credentials (client identifier and secret) from Faji to be used with Faji's OAuth-enabled API.

# OAuth example

---

- After Jane clicks Continue, Beppa requests from Faji a set of temporary credentials.
- At this point, the temporary credentials are not resource-owner-specific, and can be used by Beppa to gain resource owner approval from Jane to access her photos.



# OAuth example

---

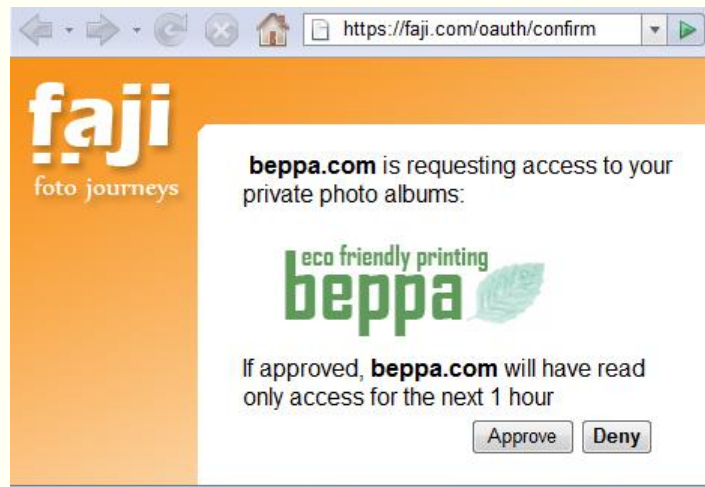
- When Beppa receives the temporary credentials, it redirects Jane to the Faji OAuth User Authorization URL with the temporary credentials and asks Faji to redirect Jane back once approval has been granted to <http://beppa.com/order>.
- Jane has been redirected to Faji and is requested to sign into the site. OAuth requires that servers first authenticate the resource owner, and then ask them to grant access to the client.



# OAuth example

---

- After successfully logging into Faji, Jane is asked to grant access to Beppa, the client. Faji informs Jane of who is requesting access (in this case Beppa) and the type of access being granted. Jane can approve or deny access.

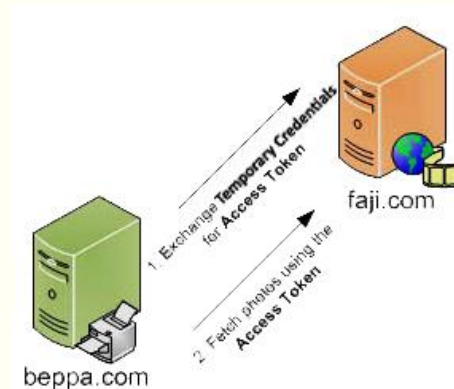




# OAuth example

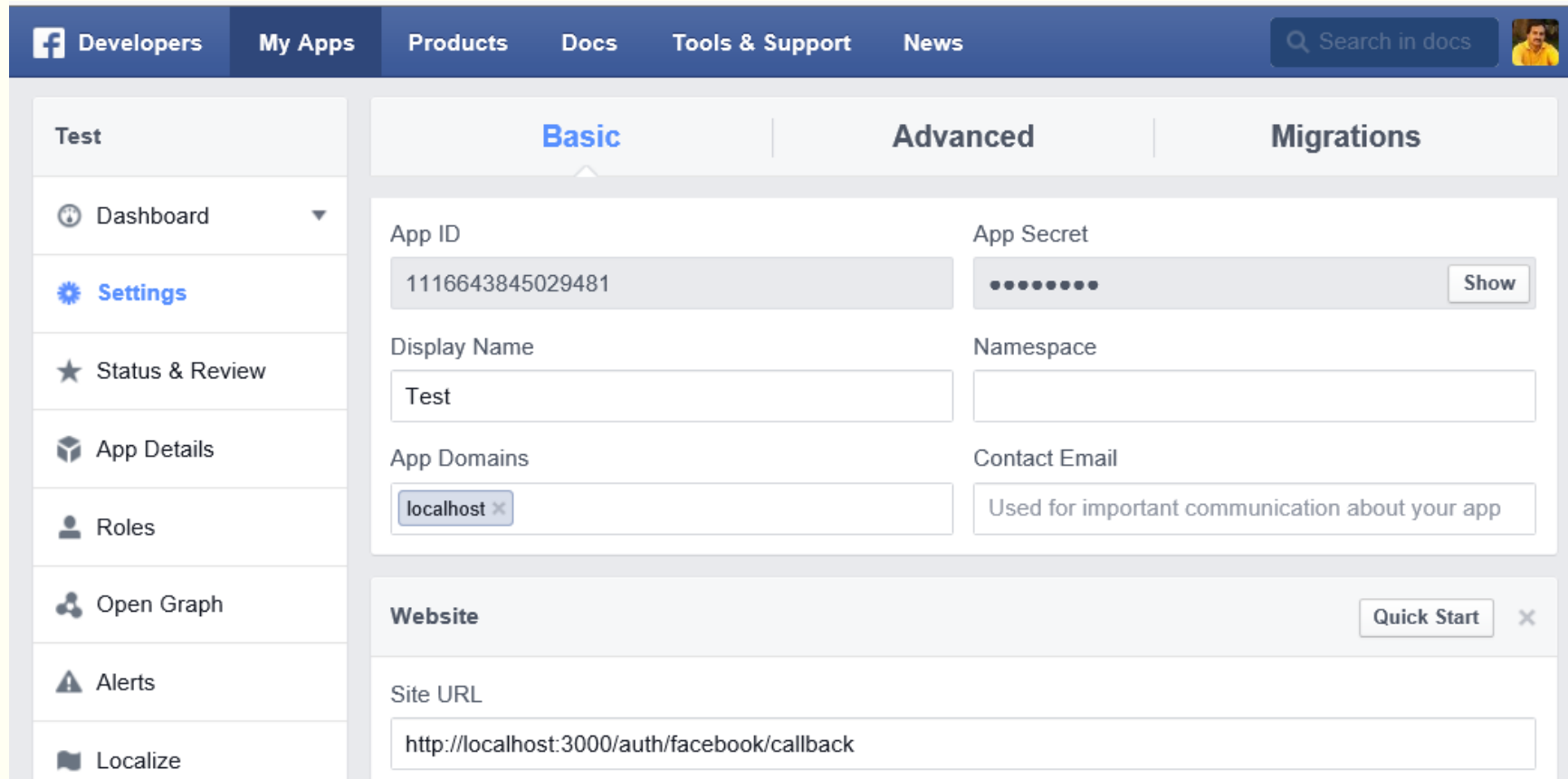
---

- Beppa uses the authorized Request Token and exchanges it for an Access Token. Request Tokens are only good for obtaining User approval, while Access Tokens are used to access Protected Resources, in this case Jane's photos. In the first request, Beppa exchanges the Request Token for an Access Token and in the second request gets the photos.



# Implementing Facebook authentication

- Register a new Facebook app. Note of the App ID and App Secret. Since we're developing locally, set App Domains as localhost. Specify Site URL



The screenshot shows the Facebook Developers 'My Apps' page. The top navigation bar includes 'Developers', 'My Apps', 'Products', 'Docs', 'Tools & Support', and 'News'. A search bar and a user profile picture are also present. The left sidebar contains a 'Test' section with links to 'Dashboard', 'Settings', 'Status & Review', 'App Details', 'Roles', 'Open Graph', 'Alerts', and 'Localize'. The main content area is divided into three tabs: 'Basic', 'Advanced', and 'Migrations'. The 'Basic' tab is active, showing the following fields:

Basic	
App ID	App Secret
1116643845029481	..... <a href="#">Show</a>
Display Name	Namespace
Test	
App Domains	Contact Email
localhost x	Used for important communication about your app

Below the 'Basic' tab, there is a 'Website' section with a 'Quick Start' button and a close icon. The 'Site URL' field is filled with 'http://localhost:3000/auth/facebook/callback'.

# Implementing Facebook Authentication

---

```
var passport = require('passport');
var FacebookStrategy = require('passport-facebook').Strategy;

var FACEBOOK_APP_ID = 'appid';
var FACEBOOK_APP_SECRET = 'secret';

app.use(passport.initialize());
app.use(passport.session());

passport.use(new FacebookStrategy({
  clientID : FACEBOOK_APP_ID,
  clientSecret : FACEBOOK_APP_SECRET,
  callbackURL : 'http://localhost:3000/auth/facebook/callback'
}, function(accessToken, refreshToken, profile, done) {
  process.nextTick(function() {
    // Assuming user exists
    done(null, profile);
  });
}));
```

# Implementing Facebook Authentication

---

```
app.get('/auth/facebook', passport.authenticate('facebook'));

app.get('/auth/facebook/callback', passport.authenticate('facebook', {
  successRedirect : '/success',
  failureRedirect : '/error'
}));

app.get('/success', function(req, res, next) {
  res.send('Successfully logged in.');
```

```
});

app.get('/error', function(req, res, next) {
  res.send("Error logging in.");//
});

app.get('/', function(req, res, next) {
  console.log("Called ")
  res.redirect('./login.html');
});
```

# Grunt

---

- Grunt.js is a Node.js JavaScript task runner that helps you perform repetitive tasks such as minification, compilation, unit testing or linting.



# Why use a task runner?

---

- Enables to write consistent code
- Maintain coding standards within teams
- Automate build process
- Automate testing and deployment and release process

# Setting Up

---

- The first thing to do in order to use Grunt is to set up Node.js.
- `npm install -g grunt-cli`
- To make sure Grunt has been properly installed, you can run the following command:
- `grunt --version`

## Next Steps

---

- The next step is to create a **package.json** and a **gruntfile.js** file in the root directory of your project.
- Creating the package.json File and execute [ npm install ]

```
"devDependencies" : {  
  "grunt" : "~0.4.0",  
  "grunt-contrib-cssmin": "*",  
  "grunt-contrib-sass": "*",  
  "grunt-contrib-uglify": "*",  
  "grunt-contrib-watch": "*",  
  "grunt-cssc": "*",  
  "grunt-htmlhint": "*",  
  "matchdep": "*"   
}
```





# Creating the gruntfile.js File

---

- Gruntfile.js is essentially made up of a wrapper function that takes grunt as an argument.

```
/**
 * Grunt file
 */
module.exports = function(grunt){

    // files to be minified and combined
    var cssFiles = [
        'public/css/file1.css',
        'public/css/file2.css'
    ];

    grunt.initConfig({
        // read the package.json
        // pkg will contain a reference to out package.json file use of which we will see later
        pkg: grunt.file.readJSON('package.json'),
```

# Creating the gruntfile.js File

---

```
// configuration for the cssmin task
// note that this syntax and options can found on npm page of any grunt plugin/task
cssmin: {
  // options for css min task
  options:{
    // banner to be put on the top of the minified file
    // using package name and todays date
    // note that we are reading our project name using pkg.name i.e name of our project
    banner: '/*! <%= pkg.name %> <%= grunt.template.today("yyyy-mm-dd") %> */\n'
  },
  combine: {
    // options for combining files
    // we have defined cssFiles variable to hold our file names at the top
    files: {
      // here key part is output file which will our <package name>.min.css
      // value part is set of input files which will be combined/minified
      'public/css/<%= pkg.name %>.min.css': cssFiles
    }
  }
}
```

# Creating the gruntfile.js File

---

```
// Load the plugin that provides the "cssmin" task.
grunt.loadNpmTasks('grunt-contrib-cssmin');

// Default task(s).
grunt.registerTask('default', ['cssmin']);

// cssmin task
grunt.registerTask('buildcss', ['cssmin']);

};
```

- `grunt.loadNpmTasks` - This is where we load the task/plugin.
- `grunt.registerTask` - This is where we register the task.
- `grunt`
  - Runs default grunt task defined with `grunt.registerTask('default', [['cssmin','uglify']])`
- `grunt buildcss`
  - Runs custom grunt task defined with `grunt.registerTask('buildcss', ['cssmin'])`

# JSHint

---

- JSHint is a program that flags suspicious usage in programs written in JavaScript.
- JSHint comes with a default set of warnings but it was designed to be very configurable.
- There are three main ways to configure your copy of JSHint:
  - you can either specify the configuration file manually via the `--config` flag,
  - use a special file `.jshintrc`
  - or put your config into your projects `package.json` file under the `jshintConfig` property

# JSHint Options

---

- Enforcing options
- When set to true, these options will make JSHint produce warnings about your code.

<u><b>camelcase</b></u>	<b>This option allows you to force all variable names to use either camelCase style or UPPER_CASE with underscores.</b>
<u>bitwise</u>	This option prohibits the use of bitwise operators such as ^ (XOR),   (OR) and others. Bitwise operators are very rare in JavaScript programs and quite often & is simply a mistyped &&.
<u>curly</u>	This option requires you to always put curly braces around blocks in loops and conditionals.
<u>equeqeq</u>	This options prohibits the use of == and != in favor of === and !==

# JSHint Options [More on <http://jshint.com/docs/options/>]

<u>funcscope</u>	This option suppresses warnings about declaring variables inside of control structures while accessing them later from the outside. Even though JavaScript has only two real scopes—global and function—such practice leads to confusion among people new to the language and hard-to-debug bugs. This is why, by default, JSHint warns about variables that are used outside of their intended scope
<u>globals</u>	Setting an entry to true enables reading and writing to that variable. Setting it to false will trigger JSHint to consider that variable read-only.
<u>maxcomplexity</u>	This option lets you control cyclomatic complexity throughout your code. Cyclomatic complexity measures the number of linearly independent paths through a program's source code
<u>maxdepth</u>	This option lets you control how nested do you want your blocks to be

## "jshintConfig": config in package.json

---

```
"jshintConfig": {  
  "bitwise": true,  
  "camelcase": true,  
  "curly": false,  
  "expr": true,  
  "eqeqeq": true,  
  "immed": true,  
  "indent": 4,  
  "latedef": "nofunct",  
  "newcap": true,  
  "undef": true,  
  "unused": true,  
  "strict": true,  
  "globalstrict": true,  
  "trailing": true,  
  "maxparams": 4,  
  "maxdepth": 2,  
  "maxcomplexity": 6,  
  "node": true,  
  "browser": true,  
  "jquery": true,
```

```
  "globals": {  
    "moment": true,  
    "before": true,  
    "describe": true,  
    "expect": true,  
    "it": true  
  }  
}
```

# Gruntfile Config

---

```
grunt.initConfig({
  // read the package.json
  // pkg will contain a reference to out package.json file
  pkg: pkg ,
  // JSHint
  jshint: {
    // JSHint configuration is read from packages.json
    options: pkg.jshintConfig,
    all: [
      'Gruntfile.js',
      'public/app/scripts/**/*.js',
      'test/**/*.js'
    ]
  },
  // Load the plugin that provides the "cssmin" task.
  grunt.loadNpmTasks('grunt-contrib-cssmin');

  grunt.loadNpmTasks('grunt-contrib-jshint');

  // Default task(s).
  grunt.registerTask('default', ['cssmin', 'jshint']);
```



# Simple Mocha

---

- Let's configure Grunt to run these tests for us automatically.
- We'll begin by installing the grunt-simple-mocha plugin using the --save-dev flag to keep package.json up to date.
  - `npm install grunt-simple-mocha --save-dev`
    - `"grunt-simple-mocha": "^0.4.0"`
    - Configure gruntfile.js

```
simplemocha: {  
  options: {  
    globals: ['expect'],  
    timeout: 3000,  
    ignoreLeaks: false,  
    ui: 'bdd',  
    reporter: 'tap'  
  },  
  all: { src: ['test/*.js'] }  
},
```

## Configure gruntfile.js and run test

---

```
// Load the plugin that provides the "cssmin" task.
grunt.loadNpmTasks('grunt-contrib-cssmin');

grunt.loadNpmTasks('grunt-contrib-jshint');

grunt.loadNpmTasks('grunt-simple-mocha');

// Default task(s).
grunt.registerTask('default', ['simplemocha', 'cssmin', 'jshint']);
```

```
G:\ITR2_BACKBONE_WS\gruntOne>grunt simplemocha
Running "simplemocha:all" (simplemocha) task
1..4
ok 1 Vehicle Test Fuel Cost should be 75.00
ok 2 Vehicle Test Fuel Cost should be 54.50
ok 3 Array indexOf() Negative Flow should return -1 when the value is not present
ok 4 Array indexOf() Main Flow should return value when the value is present
# tests 4
# pass 4
# fail 0
```