

# Introducing Node.js



# Introducing the Node.js processing model



# What is Node.js?

- The highly performant, open source *Google V8 JavaScript engine* implemented as a server-based development platform focused on non-blocking I/O intensive applications
  - created by Ryan Dahl (<https://github.com/ry>)
  - major support from Joyent (<http://www.joyent.com>).
- The project and its many supporters provide modules for:
  - file and network I/O (file system, HTTP, FTP, socket, etc.)
  - parsing (JSON, XML, CSV, URL, binary, etc.)
  - development tools (debugger, frameworks, etc.)
  - much more ...

<http://nodejs.org/api/>

<https://github.com/joyent/node/wiki/modules>

# How does Node.js work?

- Node.js provides an event-driven, non-blocking I/O model
  - Single threaded event loop executing JavaScript code
  - Functions that reach outside the engine for Input/Output require a callback function
- So, the engine never idles
  - Node.js continually moves to its next stacked task
  - When a function requiring I/O completes, its callback function is placed on the stack, and handled in due course
- Node.js is written in C and is itself multi-threaded
  - It can be extended using C modules, which can also be multi-threaded
  - JavaScript, by intentional design, always runs in the single threaded event loop

# Does Node.js scale?

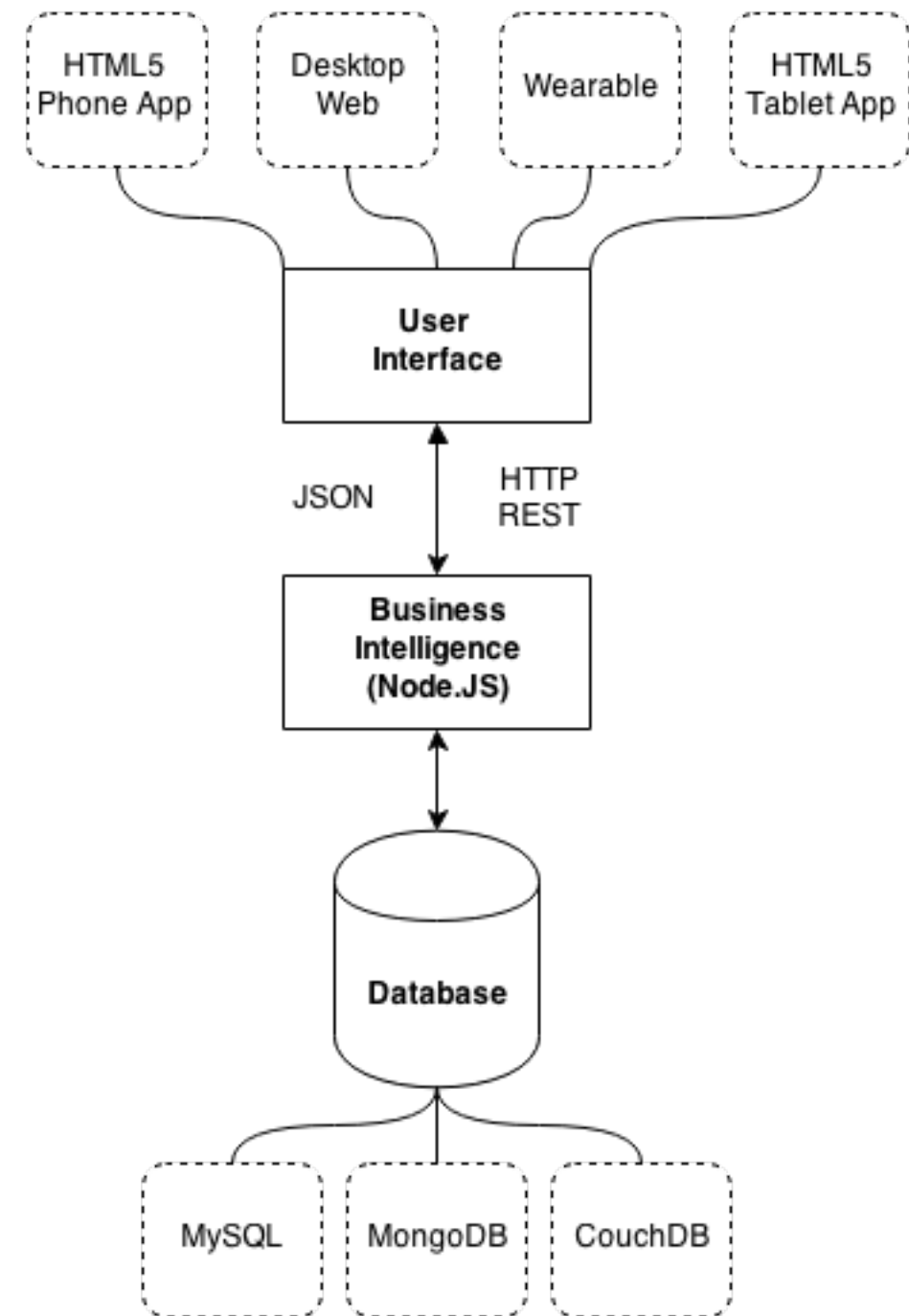
- Due to the event-driven, non-blocking I/O approach, disk and network intensive applications scale well on Node.js, particularly in relation to their relative ease of development
  - LinkedIn.com estimates 20x gain in move from Ruby on Rails to Node.js
  - <http://highscalability.com/blog/2012/10/4/linkedin-moved-from-rails-to-node-27-servers-cut-and-up-to-2.html>

# When would you not use Node.js?

- Node.js optimizes and simplifies I/O intensive applications
- So, it is not well suited for CPU heavy applications:
  - complex sorting
  - data transformation, etc.

# When would you consider using Node.js?

- Node.js optimizes and simplifies I/O intensive applications. So, it is well suited for:
  - HTTP/HTTPS JSON and XML APIs
  - Socket communications
  - Mobile / Single Page applications
- [http://nodeguide.com/convincing\\_the\\_boss.html](http://nodeguide.com/convincing_the_boss.html)



# Installing and running Node.js



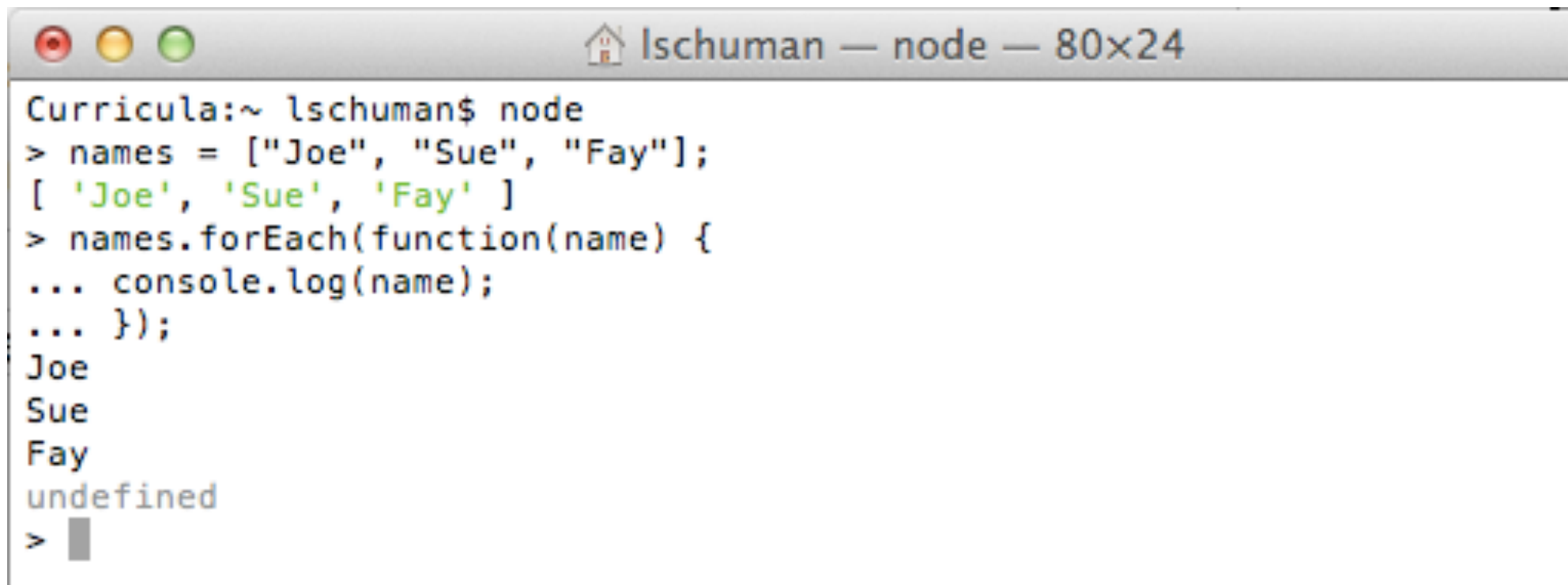


# How do I install Node.js?

- Node.js installers and/or binaries are available for Windows, Mac, Linux, and SunOS
  - <http://nodejs.org/download/>
- Source code is also freely available on the Node.js project site, or GitHub
  - <https://github.com/joyent/node>

# How do I run Node.js code?

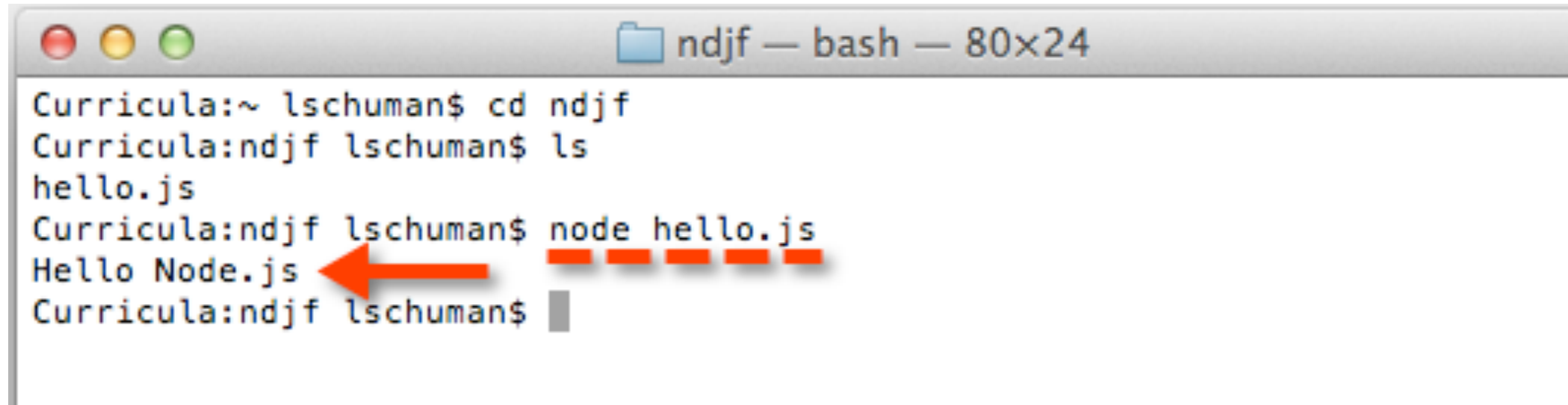
- Node.js supports a REPL (read-eval-print) loop, like Python and Ruby, enabling single or multi-line code blocks to be entered and evaluated at the command line
- launch REPL mode by entering the *node* command

A screenshot of a macOS terminal window titled "lschuman — node — 80x24". The terminal shows the Node.js REPL loop. The user enters the command "node" at the "Curricula:~ lschuman\$" prompt. Then, they enter a multi-line code block: "> names = ['Joe', 'Sue', 'Fay'];", "[ 'Joe', 'Sue', 'Fay' ]", "> names.forEach(function(name) {", "... console.log(name);", "... });". The REPL outputs "Joe", "Sue", and "Fay" on separate lines. After the code block, it outputs "undefined". The prompt ">" is followed by a cursor.

```
Curricula:~ lschuman$ node
> names = ["Joe", "Sue", "Fay"];
[ 'Joe', 'Sue', 'Fay' ]
> names.forEach(function(name) {
... console.log(name);
... });
Joe
Sue
Fay
undefined
> █
```

# How do I run Node.js code?

- JavaScript files written for Node.js can also be run using the *node* command

A screenshot of a macOS terminal window titled 'ndjf — bash — 80x24'. The terminal shows a user navigating to a directory and running a Node.js script. The output of the script is displayed on the next line. An orange arrow points from the word 'node' in the command to the output 'Hello Node.js'.

```
Curricula:~ lschuman$ cd ndjf
Curricula:ndjf lschuman$ ls
hello.js
Curricula:ndjf lschuman$ node hello.js
Hello Node.js
Curricula:ndjf lschuman$
```

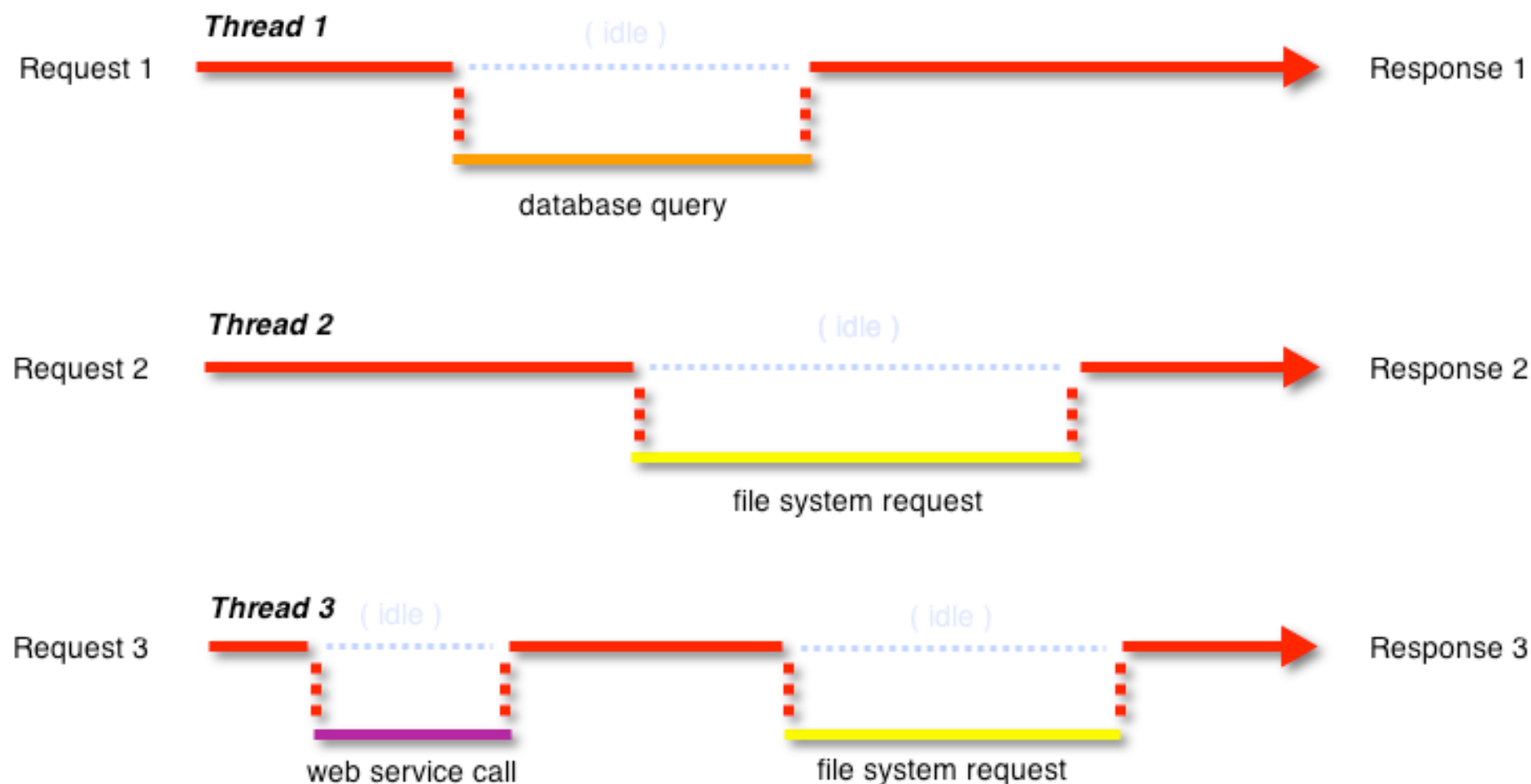
# Understanding blocking and callbacks



# Understanding single threaded request handling

- LAMP stack systems spawn a new processor thread to handle each incoming request
- The thread is idle during external I/O events, such as database queries, file system requests, external service requests, and so on.

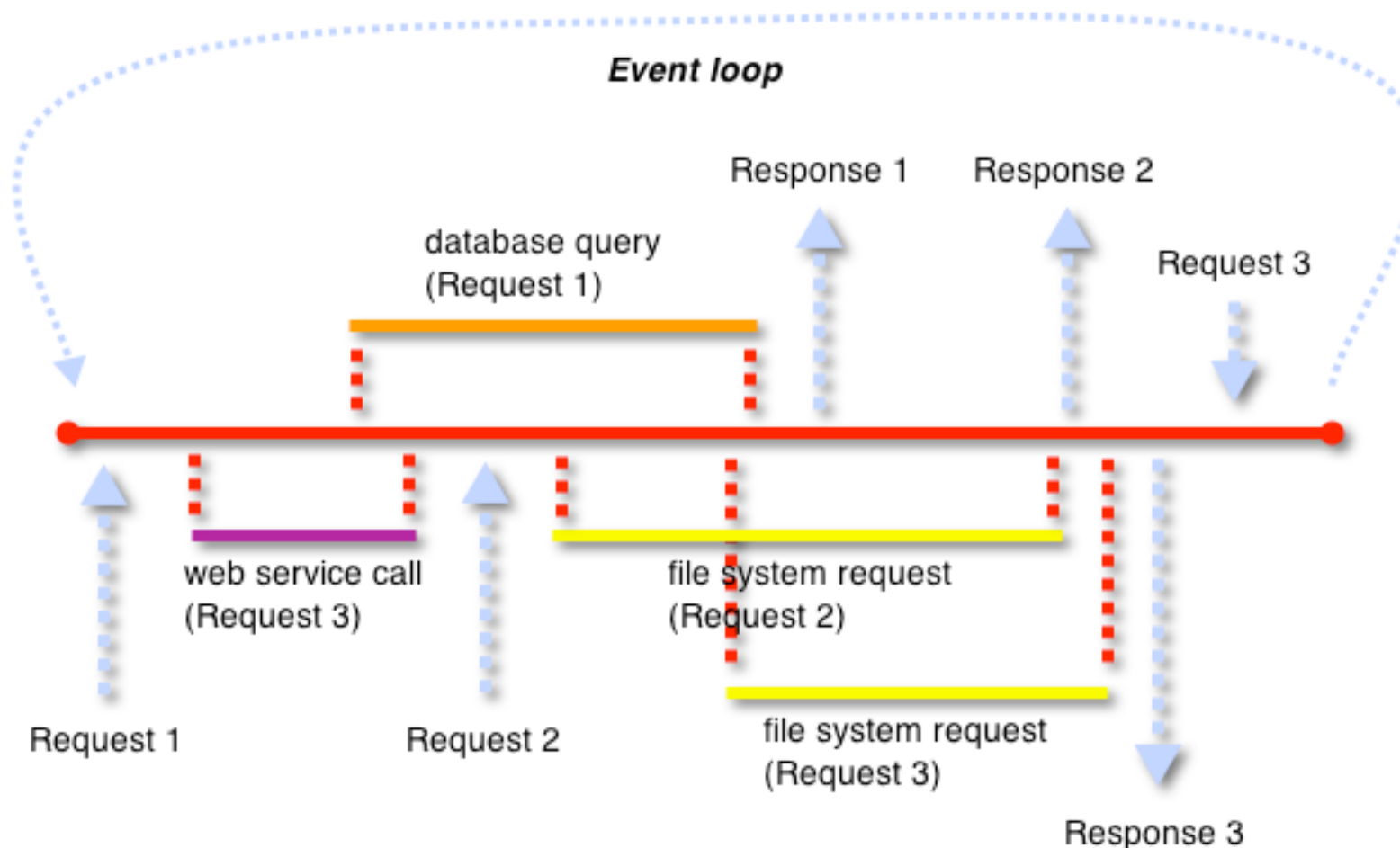
Threaded Requests Model: one thread per request (LAMP stack, etc.)



# Understanding single threaded request handling

- By contrast, Node.js relies on a single threaded event loop
  - The thread is never idle, but continues during external I/O operations
- When external operations complete, their callback function is sequenced for handling as a subsequent event in the loop

Single Threaded Event Loop Model: Node.js



# Understanding the importance of callbacks

- Node.js can still be blocked
  - The event-driven model works because callbacks are provided for long operations - generally external I/O - which would otherwise hold up the event loop
- If a long operation provides no callback, the entire system waits for it to complete before moving to the next event
- Remember, though, that Node.js is only single threaded for JavaScript code
  - The core Google V8 JavaScript engine is multi-threaded
  - Node.js modules written in C may also use multi-threading

# Coding for Node.js





# Introducing Node.js coding

- Node.js applications are (primarily) written using JavaScript
  - Surveying JavaScript is outside the scope of this training
- But, two Node.js extensions will be introduced here:
  - Including modules using the Node.js *require()* command
  - Assigning event callbacks using the Node.js *on()* method

# Making Node.js modules available for use

- A Node.js module is a reusable unit of functionality
  - Node.js modules are written to the *CommonJS* standard (<http://commonjs.org>)
- Modules are made available to Node.js code using the *require(module)* command.
  - Node.js relies on the *RequireJS* module and package loader (<http://requirejs.org>)

```
// load the built-in http module for use
var http = require("http");
var server = http.createServer();
```

*Note: module creation and inclusion are discussed further throughout the course*

# Assigning event callbacks

- Node.js modules implement event dispatching by prototyping the *EventEmitter* class
- Objects of this class expose methods including:
  - *emitter.addListener(event, listener)*
  - *emitter.on(event, listener)*
  - *emitter.once(event, listener)*
  - *emitter.removeListener(event, listener)*
- <http://nodejs.org/docs/latest/api/events.html#emitter.on>

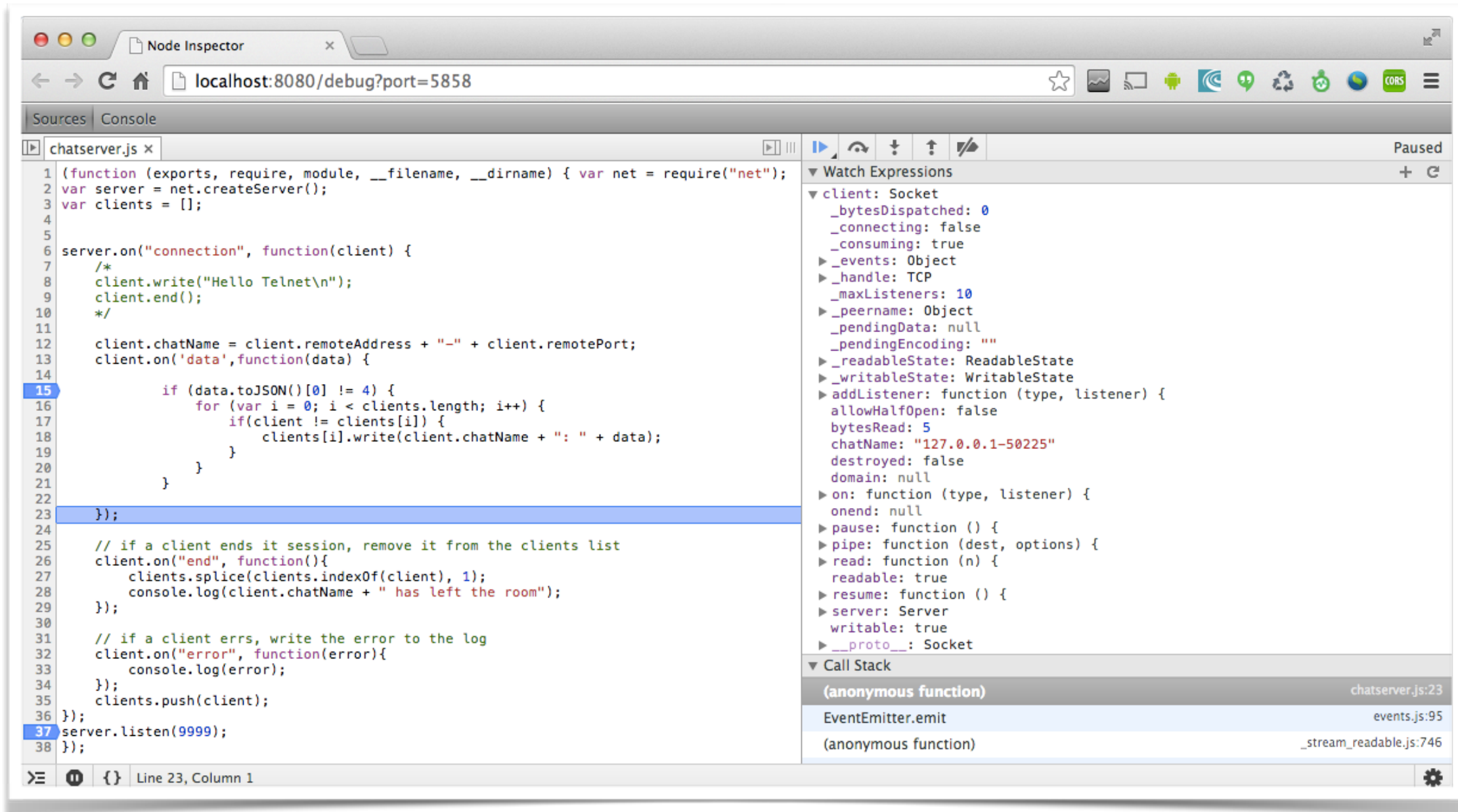
// For example, to assign an anonymous function to handle network connection events

```
var net = require("net");
var server = net.createServer();
server.on("connection", function(client) {
  client.write("Hello Node.js");
})
```

*Note: callback assignment is discussed further throughout the course*

# Debugging Node.js Applications

- Node Inspector



# Node Inspector

- You can install node inspector by typing the following at a Command prompt:

```
npm install -g node-inspector
```

- Start the debugging process by issuing the following command:

```
node-debug app.js
```

- Chrome or Opera must be your default browser
- Your browser will automatically launch when your app starts up
- Set breakpoints by right-clicking on a line number in the debugger.

# Demo I: Creating a socket based chat server

- In this exercise you will create a simple telnet chat server using the built-in networking library, suppress local message echo, and handle errors
- After completion, you should be able to:
  - Include a built-in Node.js library
  - Create a network server object listening to a specified port
  - Create client objects to manage each server connection
  - Handle network connection, data, end, and error events

# Introducing Node.js modules

- Node.js functionality is organized into units called *modules*
- A *module* is JavaScript or C from which functions have been exported for external use
  - Several modules are built into the Node.js binary (core modules)
  - Many others are freely available for inclusion (file modules)
- Modules also serve as a primary organizational approach for custom Node.js code

# Surveying built-in Node.js modules

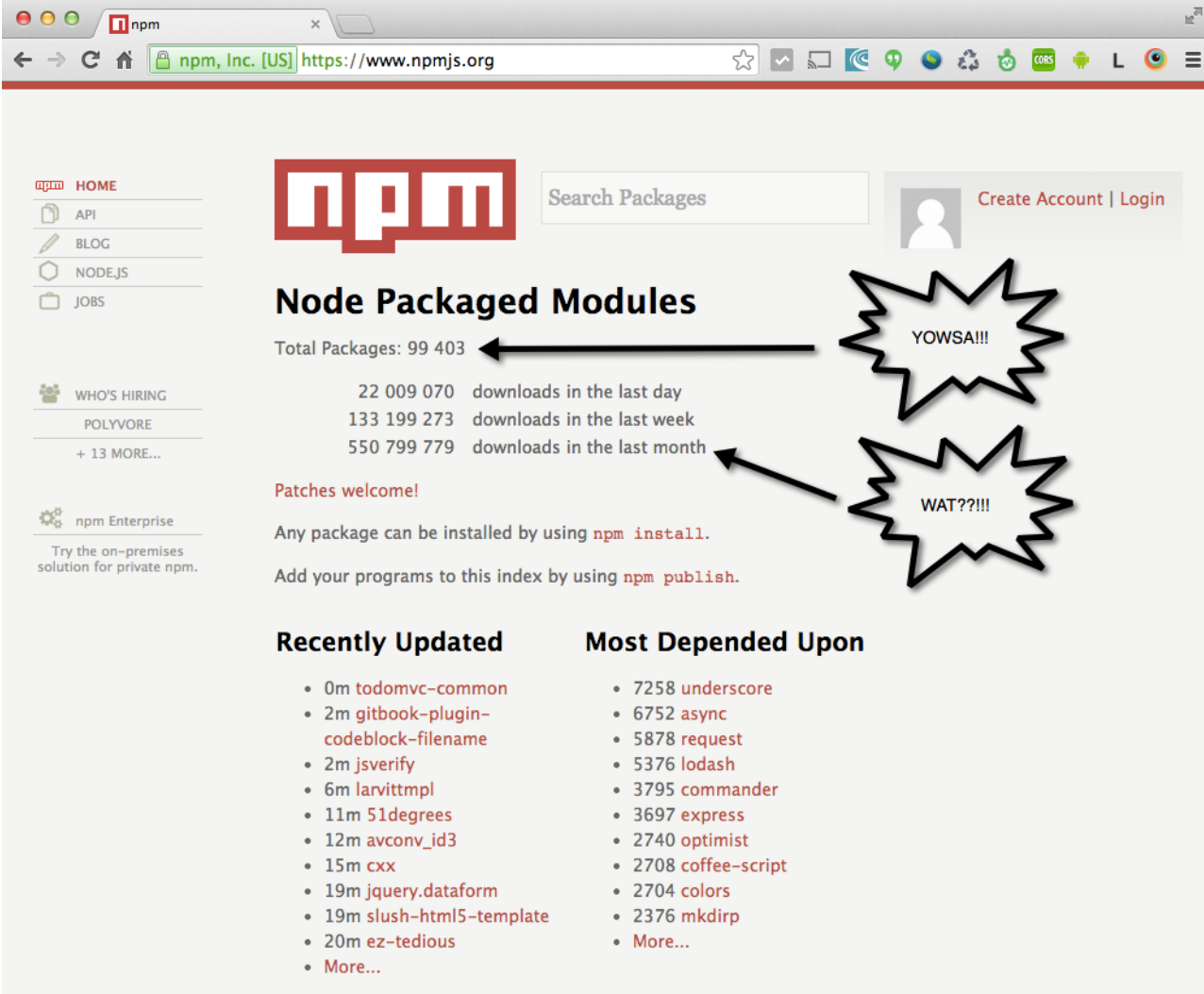
- Node.js contains built-in modules including the following
  - <http://nodejs.org/api/>

|        |      |        |      |             |                 |
|--------|------|--------|------|-------------|-----------------|
| net    | http | https  | url  | querystring | url             |
| crypto | fs   | buffer | path | util        | <i>more ...</i> |



# Surveying community Node.js modules

- Hundreds of community supported modules include support for:
  - Relational database access (MySQL, PostgreSQL, Oracle, MS SQL Server, more ... )
  - Document database access (MongoDB, CouchDB, Hive, Redis, more ... )
  - OpenSSL, crypto, hashing, parsing (JSON, XML, CSV, YAML, more ... )
  - Sound, video, and graphics processing
  - Web frameworks and templating
- <https://www.npmjs.org>



The screenshot shows the npm website interface. At the top, there's a navigation bar with the npm logo, a search bar, and links for 'Create Account' and 'Login'. Below the navigation bar, the main heading is 'Node Packaged Modules'. To the right of this heading, there are two starburst graphics: one saying 'YOWSA!!!' and another saying 'WAT??!!'. The main content area displays statistics: 'Total Packages: 99 403', '22 009 070 downloads in the last day', '133 199 273 downloads in the last week', and '550 799 779 downloads in the last month'. Below the statistics, there's a section for 'Patches welcome!' and instructions on how to install and publish packages. At the bottom, there are two columns: 'Recently Updated' and 'Most Depended Upon', each listing several popular packages with their update or dependency counts.

| Recently Updated                       | Most Depended Upon   |
|--|----------------------|
| • 0m todomvc-common                    | • 7258 underscore    |
| • 2m gitbook-plugin-codeblock-filename | • 6752 async         |
| • 2m jsverify                          | • 5878 request       |
| • 6m larvittmpl                        | • 5376 lodash        |
| • 11m 51degrees                        | • 3795 commander     |
| • 12m avconv_id3                       | • 3697 express       |
| • 15m cxx                              | • 2740 optimist      |
| • 19m jquery.dataform                  | • 2708 coffee-script |
| • 19m slush-html5-template             | • 2704 colors        |
| • 20m ez-tedious                       | • 2376 mkdirp        |
| • More...                              | • More...            |

# Using Node.js modules



# Understanding the `require()` function

- A module is included in a file using `require("module-name")`

```
var http = require("http");  
var server = http.createServer();
```

- Node.js looks for the module in this order:
  1. by name, as a core module compiled with Node.js from its source */lib* folder
  2. by name in the local folder, with either of these extensions:
    - .js - interpreted JavaScript
    - .node - compiled C
  3. by an absolute ( `"/.."` ) or relative ( `"./.."` or `"../.."` ) path in the `require()` function
  4. in a `node_modules` folder in the current folder, or successive parent folders up to the root
- If not found, an error is thrown: `MODULE_NOT_FOUND`
- Additional module packaging options are documented and available
  - <http://nodejs.org/api/modules.html>

# Using the HTTP module

- The HTTP module is designed to
  - ease use of lower-level features of the HTTP protocol, and
  - enable streaming transfer and parsing of large, possibly chunk-encoded messages
- <http://nodejs.org/api/http.html>

# Creating and implementing an HTTP server

- The HTTP module enables creation of a server to listen for and respond to HTTP requests on a port, using:
  - `createServer(callback)`
  - `listen(port-number)`

```
var http = require("http");  
var server = http.createServer(requestHandler);  
function requestHandler(request, response) {  
    ...  
}  
server.listen(7777);
```

- Because of the nature of JavaScript, the same behavior could be built with chained statements

```
var server = require("http").createServer(function(request, response) {  
    ...  
}).listen(7777);
```

# Writing HTTP response headers and content

- Request headers are exposed by the *request* argument passed to the request handler, and response headers and content are written to the *response* object
- The response object exposes methods including:
  - *writeHead(status-code, header-object)* - return HTTP code with specified headers
  - *write(text)* - write content to the response body
  - *end()* - return the response to the requestor
- HTTP headers are represented as JavaScript objects, like:

```
{  
  "content-type": "text/plain",  
  "connection": "keep-alive",  
  "content-length": "777",  
  "accept": "*/*"  
}
```

# Organizing code as modules

- Node.js supports both interpreted JavaScript and compiled C modules, and has adopted the CommonJS API for writing application extensions in JavaScript.
- <http://www.commonjs.org>
- JavaScript modules, written to this standard, are a primary packaging and organizational approach for Node.js code
- CommonJS modules support:
  - dependency management
  - scope isolation
  - relative module identifiers

# Writing a Node.js module

- Any JavaScript file may be imported to another in Node.js by using the *require()* function

caller.js

-----

```
// require module via relative path to a file in the same folder as caller  
var my_module = require("./my_module.js");
```

- Code imported in this way is encapsulated, except for values assigned to an *exports* object which is injected into this code at runtime by the *require()* function
- Variables and functions are exported from a module by assigning them to properties of the *exports* object

my\_module.js

-----

```
function foo(bar) {  
    return bar + " Sinister";  
}  
exports.foo = foo;
```



# Using a Node.js module

- Exported variables and functions are accessible through the module reference created by the *require()* function

```
caller.js
-----
// require module via relative path to a file in the same folder as caller
var my_module = require("./my_module.js");
var value = my_module.foo("Simon");

console.log(value); // displays "Simon Sinister";
```

# Using the URL module

- The *url* module parses URL strings to/from objects
  - <http://nodejs.org/api/url.html>
  - A URL would be commonly obtained from *request.url* in an HTTP request handler
- Three url module methods are available:
  - *parse(urlString [, parseQueryString, ] [slashesDenoteHost])*
  - *format(urlObject)*
  - *resolve(fromUrl, toUrl)*
- So, for example, to parse and use an incoming URL:

```
var http = require("http");
var url = require("url");
http.createServer(function(request, response) {
  var parsedUrl = url.parse(request.url);
  switch(parsedUrl.pathname) {
    case "/": {
      ...
    }
  }
});
```

# Understanding how a URL is parsed

`http://username:password@host.com:9999/my/path?query=value#anchor`

- If parsed as an object, these properties and values would be available:
  - `href: 'http://username:password@host.com:9999/my/path?query=value#hash'`
  - `protocol: 'http'`
  - `host: 'host.com:9999'`
  - `auth: 'username:password'`
  - `hostname: 'host.com'`
  - `port: '9999'`
  - `pathname: '/my/path'`
  - `search: '?query=value'`
  - `path: '/my/path?query=value'`
  - `query`
    - `url.parse(url, true): {'query': 'value'}`
    - `url.parse(url, false): 'query=value'`
  - `hash: '#anchor'`

# Using a third party JavaScript library in Node.js

- Two key implications of using Node.js are:
  - all built-in functionality of the Google V8 JavaScript engine is available
  - third-party JavaScript libraries are easily converted for use as modules
- At the most basic level, converting a library for use in Node.js is as simple as:
  1. assign top-level functions and properties to the *exports* object, then
  2. *require()* the file

```
./lib/mylib.js
```

```
-----
```

```
function encrypt(value) { ... }  
exports.encrypt = encrypt;
```

```
caller.js
```

```
-----
```

```
var mylib = require("./lib/mylib.js");  
var encryptedText = mylib.encrypt("Something confidential ...");
```

# Parsing and using JSON data

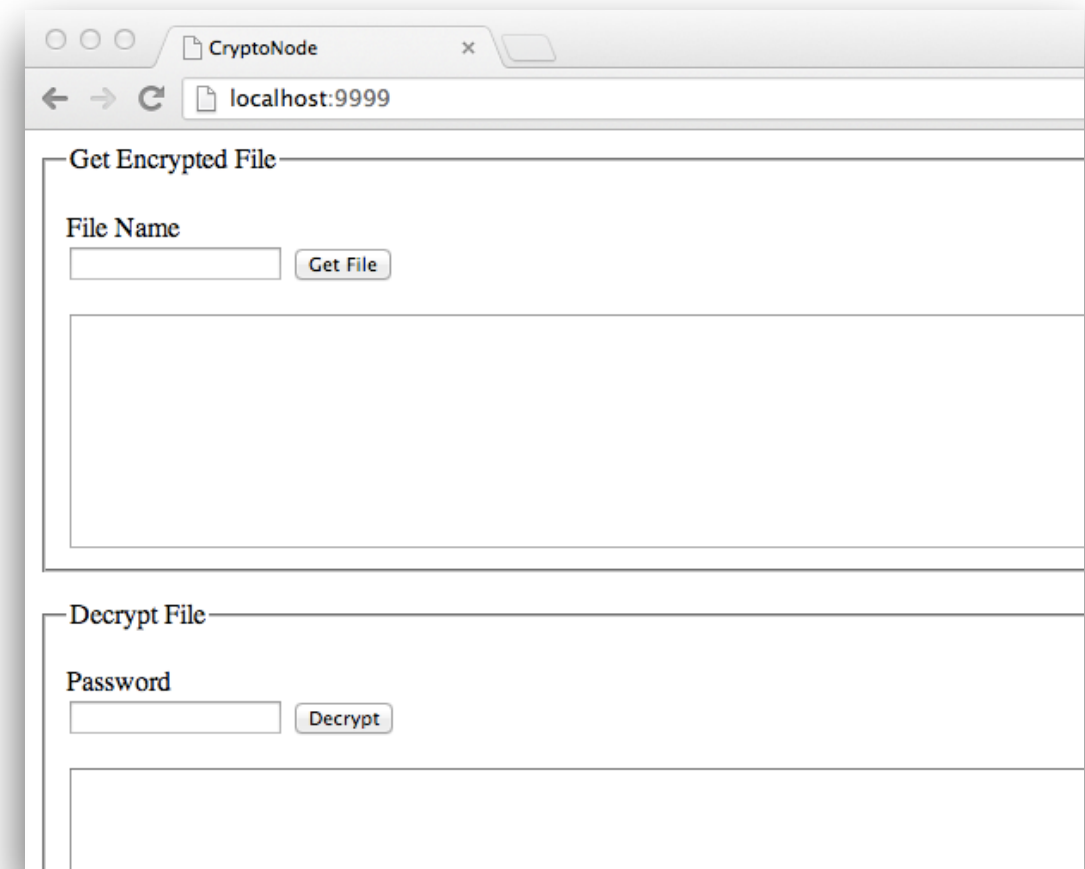
- Like any modern browser, Node.js natively supports standard JSON parsing methods:
  - *JSON.parse(JSON string)*
  - *JSON.stringify(javascript object)*
- So, for example, if a third party encryption library returns JSON as output, this can be parsed to access its values:

```
var sjcl = require("./sjcl.js");  
var encryptedData = sjcl.encrypt("Something confidential ...");  
var encryptedObject = JSON.parse(encryptedData);  
var cipherText = encryptedObject.ct;
```

- This example is taken from using the open source Stanford JavaScript Crypto Library, which returns output as a JSON formatted string, with the encrypted ("cipher") text available in the *ct* property of the object
  - <http://crypto.stanford.edu/sjcl/>

## Demo 2: Routing requests and using external libraries as modules

- In this exercise you will enable users to specify a remote file name, encrypt that file, retrieve the encrypted file, and decrypt it on the client, using the same third-party encryption library
- After completion, you should be able to:
  - create simple HTTP request routing
  - respond to HTTP requests by loading and returning a file with appropriate headers
  - configure a third party JavaScript library for use as a module
  - use the Stanford JavaScript Crypto Library on both client and server to encrypt and decrypt text files



The screenshot shows a web browser window with the title 'CryptoNode' and the address bar displaying 'localhost:9999'. The page contains two main sections:

- Get Encrypted File:** This section has a label 'File Name' above a text input field. To the right of the input field is a button labeled 'Get File'. Below the input field is a large, empty rectangular area, likely for displaying the retrieved file content.
- Decrypt File:** This section has a label 'Password' above a text input field. To the right of the input field is a button labeled 'Decrypt'. Below the input field is another large, empty rectangular area, likely for displaying the decrypted file content.

# Introducing Express

- *"Express is a minimal and flexible Node.js web application framework, providing a robust set of features for building single and multi-page, and hybrid web applications."*
- <http://expressjs.com/>
- *"[Ruby] Sinatra inspired ... [f]ast, un-opinionated, minimalist web framework for Node.js."*
- <https://npmjs.org/package/express>

# Understanding the problems Express addresses

- URL routing and file serving quickly grow complex, because each request may vary as to:
  - HTTP request methods or "verbs" (GET, POST, PUT, DELETE, etc.)
  - request data location (URL parameters or request body)
  - request content-type (text/html, text/javascript, image/jpeg, etc.)
  - need for file caching and redirection
  - need to display dynamic values in a templated view
- Express provides a flexible, straightforward approach to managing routes, files, redirection, and views, as well as an extensible (or replaceable) template system called *Jade*



# Understanding the Express design philosophy

- *"How should I structure my application?"*

*There is no true answer to this question, it is highly dependent on the scale of your application and the team involved. To be as flexible as possible Express makes no assumptions in terms of structure. Routes and other application-specific logic may live in as many files as you wish, in any directory structure you prefer."*

- <http://expressjs.com/faq.html>
- This unit provides a concise introduction to the default Express application structure

# Creating an Express.js application

- Creating a new Express.js application breaks down into three basic steps:
  1. create a new folder (*mkdir*) and navigate (*cd*) to that folder
  2. run the *express* command
    - generates local file structure and default *package.json* file
  3. run the *npm install* command
    - installs dependencies specified in *package.json* (e.g., Jade template system)

# Understanding package.json

- A *package.json* file resides in the root folder of an Express.js application, to define:
  - *name* - unique public application name, if application will be made public on npmjs.org
  - *version* - standard 0.0.1 scheme
  - *private* - true or false, stating whether package will be publicly available
  - *scripts* > *start* - default startup command (e.g., node app to run file named app.js)
  - *dependencies* - other required Node.js packaged modules to be loaded from <https://npmjs.org> during npm install of this application
- New applications include this generated *package.json* file:

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "express": "~4.0.0",
    "static-favicon": "~1.0.0",
    "morgan": "~1.0.0",
    "cookie-parser": "~1.0.1",
    "body-parser": "~1.0.0",
    "debug": "~0.7.4",
    "jade": "~1.3.0"
  }
}
```

# Introducing application configuration

- Express.js applications rely on a configured instance of an *express()* object, named *app* by default
  - Settings and features are configured through this object

```
var app = express();
var env = process.env.NODE_ENV || 'development';
if ('development' == env) {
  app.set('port', process.env.PORT || 3000);
  app.set('view engine', 'jade');
}
```

# Introducing route configuration

- Alternately, you can define and invoke routes using the Router public API

```
var homeRouter = express.Router();
  homeRouter.get('/index', function(req,res,next) {
    // do something
  });
  homeRouter.post('/index', function(req,res,next) {
    // do something
  });
app.use('/index',homeRouter);
```

In addition, you can also have Express automatically parse REST urls as illustrated by the following example:

```
app.use('/widgets/:widgetId', function(req, res, next) {
  // req.params.widgetId exists here
});
```

# Introducing page rendering

- The Express.js response object adds two methods to a standard Node.js response object.
  - `send(string)` - return arbitrary string to client
  - `response.render(template, configuration-object)` - render the specified view template using the values in the configuration object, and send the resulting string to client
- So, for example, a GET request for the root route ("/") might be handled by rendering a view template named *index* with a configuration object which provides a *title* property

```
var routes = require("./routes.js");
...
app.get("/", routes.index);
...
```

```
routes.js
-----
exports.index = function(req, res){
  res.render('index', { title: 'Hello Express!' });
};
```

# Introducing the Jade template engine

- A *view template* is a text file which is interpreted during request handling to generate the HTML or other output to be sent to the client in response
  - may represent either a whole or partial page
  - can be configured to support data object(s) injected during rendering process, to display dynamic data within a page
- By default, Express supports a template system called *Jade*
  - templates stored in a folder named */views*
  - templates stored as text files with the *.jade* extension
  - provides a shorthand syntax for defining HTML templates

# Introducing the Jade template engine

- For example, a shared page layout could be defined, with a replaceable *block* named *content*, as:

```
layout.jade
-----
doctype 5
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
```

- An individual page could extend (wrap itself within) this layout, and use configuration data passed to it as part of the *response.render(template, configuration-object)* method

```
res.render('index', { title: 'Hello Express!' });
...
```

```
index.jade
-----
extends layout

block content
  #container
    #logo
      img(src='images/logo.png')
    #display
      h2 #{title}!
```



# Demo 3: Installing Express and creating a new application

- In this exercise you will install the Express.js framework for Node.js, create a new application, review the file structure, then modify the default route view.
- After completion, you should be able to:
  - Install Express.js for global use by Node.js
  - Create and install dependencies for a new Express.js application
  - Understand the basic Express.js file structure
  - Understand the basic Express.js approach to route and view handling

# Introducing POST handling and form data access



# Accessing form data when handling POST requests

- HTTP POST requests are handled using the same *app.VERB(route, function)* syntax as GET requests
- Field values sent with a POST request are accessed by name through the *body* property of the *request* object
- For example, a Jade template might specify *username* and *password* fields in a form posting back to itself from the root of the application (e.g., a login form)

```
login-form.jade
```

```
-----
```

```
extends layout
```

```
block content
```

```
  #container
```

```
    #logo
```

```
      img(src='images/logo.png')
```

```
    #display
```

```
      form(method='post')
```

```
        div
```

```
          | Username
```

```
            input(type='text', name='username')
```

```
            input(type='text', name='password')
```

```
            input(type='submit', value='Log In')
```

# Accessing form data when handling POST requests

- The value of these fields would be accessed by name through the body property of the request object in the relevant request handler

```
var formRouter = express.Router();

formRouter.post('/login-form', function(req,res,next) {
  if(IsValid(req.body.password)) {
    var username = req.body.username || 'Default User';
    req.session.username = username;
    res.redirect('/home');
  } else {
    res.render('index', { title: 'Hello Express!' });
  }
});
```

# Introducing session management



# Creating a session aware Express.js application

- Install the Session Management Package

```
npm install express-session
```

- Enabling Session Management for the App:

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

app.use(cookieParser())
app.use(session({
  secret: 'keyboard cat',
  saveUninitialized: true, // avoids default warning message
  resave: true,           // avoids default warning message
  cookie: { secure: true }
}));
```

# Setting and accessing session data

- Session data is set and read through the *session* property of the *request* object in session-aware applications
- For example, the form post handler might assign a *username* value for a *session*, then *redirect()* to a different page

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

/* GET home page. */
router.get('/', function(req, res) {
  console.log(req.session);
  if (typeof req.session.username == 'undefined') {
    res.render('index', { title: 'Express Sessions Management' });
  } else {
    res.redirect('/home');
  }
});

router.post('/', function(req, res) {
  var username = req.body.username || "Default User";
  req.session.username = username;
  res.redirect("/");
});

module.exports = router;
```

# Introducing runtime client assets



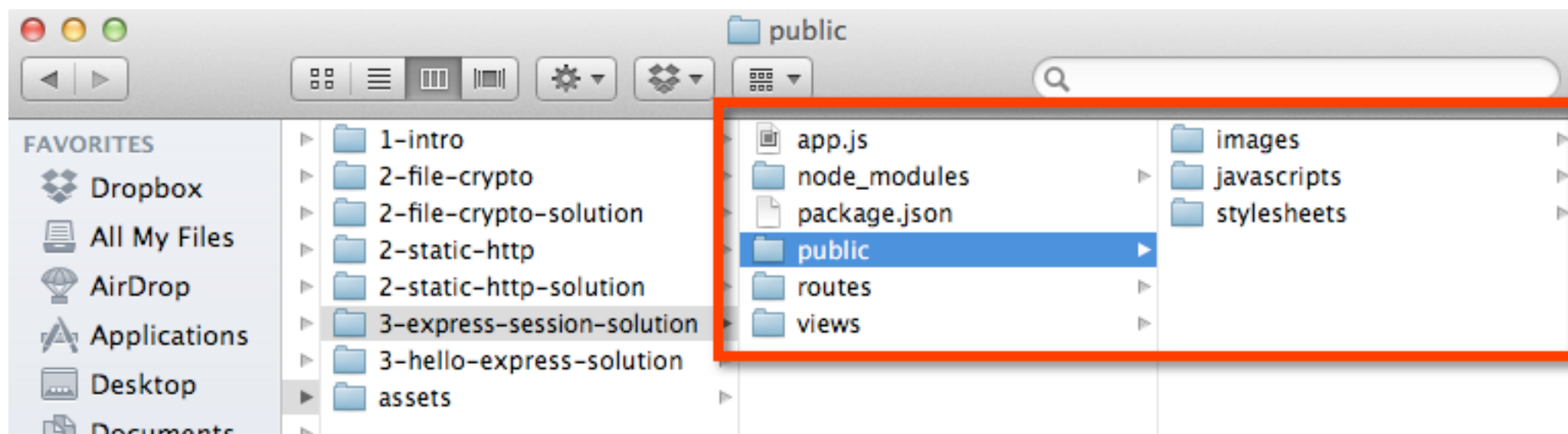


# Deploying assets for client-side use at runtime

- By default, Express.js configures itself to use a folder named */public* for static runtime assets such as images, css, and client-side Javascript

```
app.configure(function(){  
    ...  
    app.use(express.static(path.join(__dirname, 'public')));  
});
```

- This folder is configured with */images*, */javascripts*, and */stylesheets* sub-folders



# Using static assets at runtime

- Assets deployed to */public* may be used by the relevant sub-folder path in view templates
- For example, a Jade template might render *images/logo.png* as part of a page

```
index.jade
-----
extends layout

block content
  #container
    #logo
      img(src='images/logo.png')
    #display
      form(method='post')
        div
          br
          | Username
          input(type='text', name='username')
          input('type='submit', value='Log In' )
```

# Demo 4: Using Express.js session management

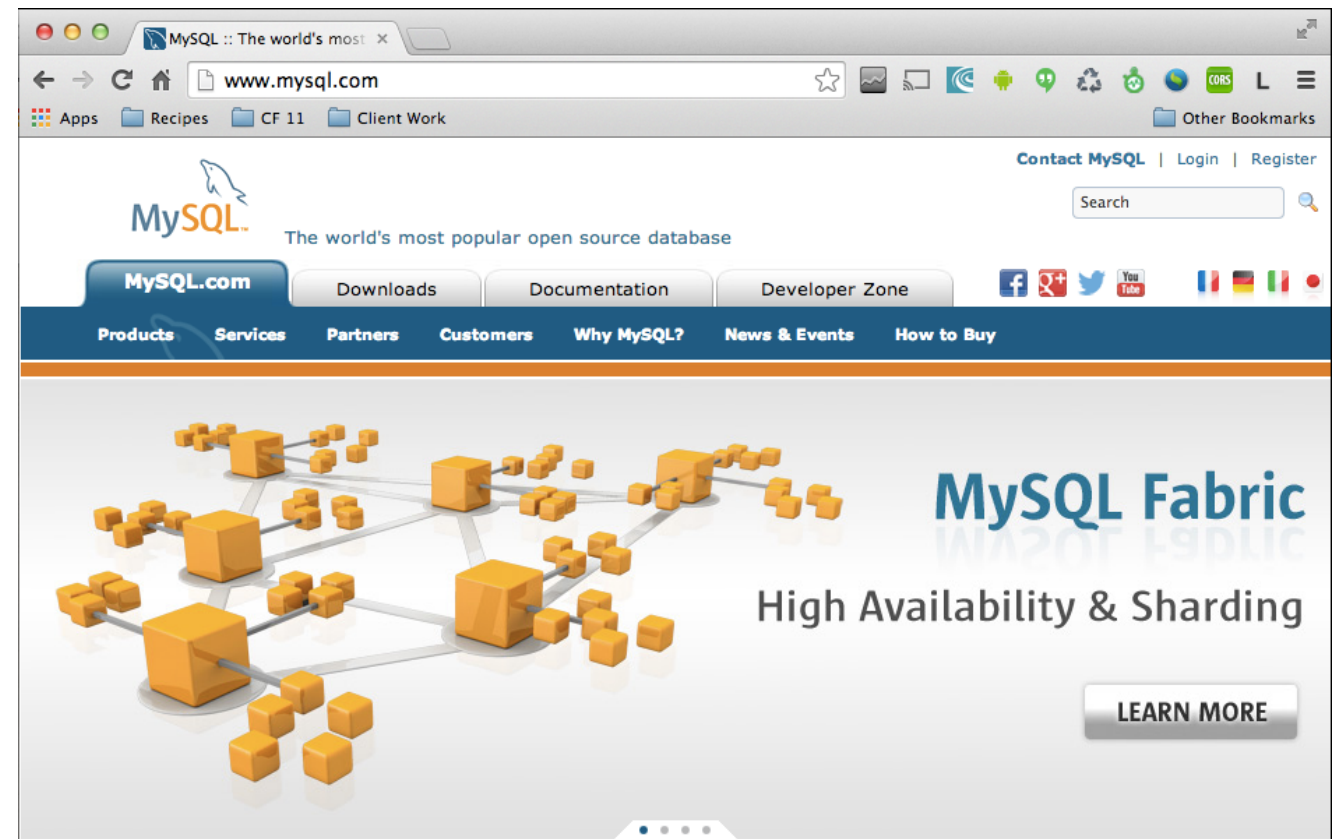
- In this exercise you will implement session management for an Express.js application, populate session data, and inject session data into a template through its configuration object.
- After completion, you should be able to:
  - Create an Express.js application with session management
  - Deploy assets for use in runtime page requests
  - Assign GET and POST route handlers
  - Access and use POST form request data in a route handler
  - Assign and access session data in a route handler
  - Inject session data into a template using its configuration object
  - Display render configuration data within a Jade template

# Working with MySQL



# Introducing MySQL

- “The World's Second Most Popular Open Source Database.”
- Owned by ORACLE corporation
- Included in xAMP stacks
- Uses the structured query language to create, read, update, and delete records in a set of relational tables.



# Introducing the MySQL Module

- An open-source node.js database driver that's written in Javascript and is available for free under the MIT license.
- Install with npm
- `npm install mysql`

# Connecting to a MySQL Database

```
var connection = mysql.createConnection({  
  host      : 'localhost',  
  database  : 'NodeJsExamples',  
  user      : 'root',  
  password  : ''  
});  
  
connection.connect();  
  
var sql = "SELECT * from Person"  
  
connection.query(sql, function(err, result, fields) {  
  if (err) throw err;  
  console.log(result);  
});  
  
connection.end();
```

# Using Connection Pooling

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

var pool = mysql.createPool({
  host      : 'localhost',
  database  : 'NodeJsExamples',
  user      : 'root',
  password  : ''
});

pool.getConnection(function(err, connection) {
  connection.query( 'SELECT * FROM foo', function(err, result) {
    connection.release();
  });
});
```



# Avoiding SQL Injection Exploits

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

# Executing Inserts and Updates

```
// quick insert:
```

```
var post  = {firstName: 'Steve', lastName: 'Drucker'};  
var query = connection.query('INSERT INTO Person SET ?', post, function(err,  
result) {  
    // Cool!!!  
});
```

```
// quick update:
```

```
var put  = {firstName: 'Steve', lastName: 'Drucker'};  
var query = connection.query('Update Person SET ? Where id= ?', [put,1],  
function(err, result) {  
    // Cool!!!  
});
```

# Getting the ID of a Newly Inserted Row

```
connection.query('INSERT INTO Person SET ?',  
    {firstName: 'Steve'},  
    function(err, result) {  
        if (err) throw err;  
  
        console.log(result.insertId);  
    })  
);
```

# Using Jade to Output Dynamic Data

```
extends layout
```

```
block content
```

```
  h1= title
```

```
  table
```

```
    thead
```

```
      tr
```

```
        th Quantity
```

```
        th Description
```

```
    tbody
```

```
      - each item in rows
```

```
        tr
```

```
          td = item.quantity
```

```
          td(style='width: '+(100/2)+'%') #{item.description}
```

```
          td= item.salePrice
```

# Demo: Creating and Outputting Dynamic SQL Queries

- Output a dynamic recordset into an HTML Table
- Implement an HTML form using Jade
- Implement server-side data validation
- Implementing a simple REST api

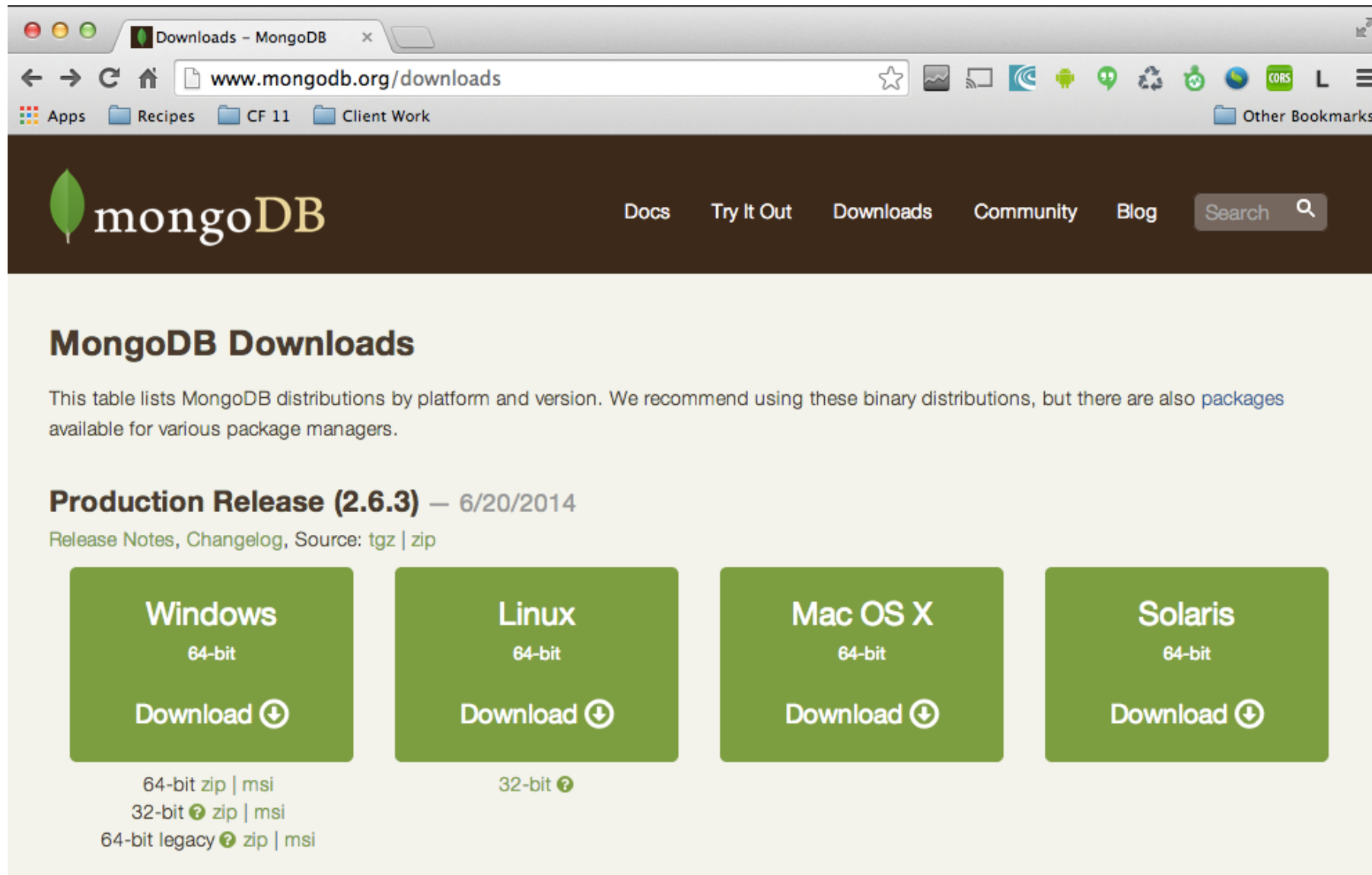
# Working with MongoDB

- An open-source document database, written in C++
- Records are similar to JSON Objects
- Command-Line Interface
- Advantages:
  - Documents correspond to native data types in javascript-based languages
  - Embedded documents and arrays reduce the need for expensive joins
  - Dynamic schema supports fluent polymorphism

```
{  
  name: "sue",  
  age: 26,  
  status: "A",  
  groups: [ "news", "sports" ]  
}
```

← field: value  
← field: value  
← field: value  
← field: value

# Installing MongoDB



- 1. Unzip the archives to a temp folder
- 2. Copy the files from the temp/mongo/bin folder to wherever you want mongo to “live”, e.g. c:\program files \mongo or Applications::Mongo

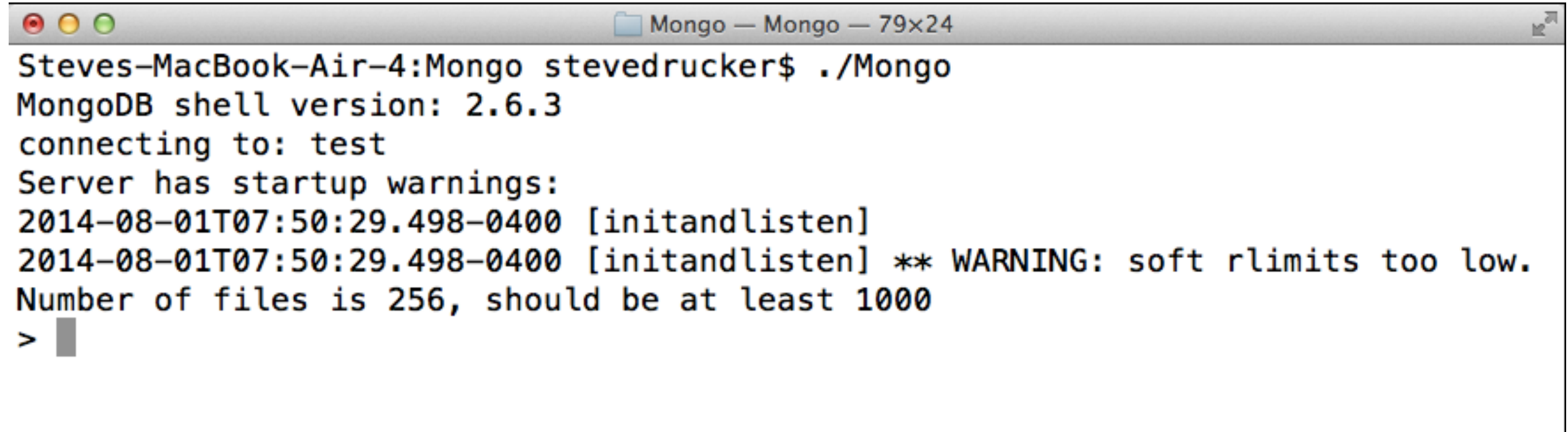
# Starting MongoDB

```
./mongod --dbpath /library/webserver/documents/ndjf/4-db-mongo/data
```





# Using the Mongo Console



```
stevedrucker$ ./Mongo
MongoDB shell version: 2.6.3
connecting to: test
Server has startup warnings:
2014-08-01T07:50:29.498-0400 [initandlisten]
2014-08-01T07:50:29.498-0400 [initandlisten] ** WARNING: soft rlimits too low.
Number of files is 256, should be at least 1000
>
```

- Connect to a database using the use command, e.g.

**use NodeJSExamples**

# Inserting New Documents

- MongoDB will automatically assign a unique identifier (id) to each record. You can override this behavior by specifying an id property for each document.
- “db” refers to the database that is currently in use (NodeJsExamples)
- Collections are analogous to RDBMS tables
- The collection “productcollection” is created on-the-fly if it did not previously exist.

```
db.productcollection.insert(  
  {  
    "title": "Jedi Lightsaber",  
    "description" : "Conquer the galaxy!",  
    "unitPrice" : 99.25  
  }  
)
```

# Updating Documents

```
db.collection.update(  
    <query>,  
    <update>,  
    {  
        upsert: <boolean>,  
        multi: <boolean>,  
        writeConcern: <document>  
    }  
)
```

# Combining Insert & Update Operation

```
db.productcollection.update(  
  { _id : ObjectId("53db8a2e85a7c57aa1715e5d") },  
  { $set : { description : 'Have fun with Cosplay ' } },  
  { upsert: true }  
)
```

# Retrieving Data

```
> db.productcollection.find().pretty()
{
  "_id" : ObjectId("53db8a2e85a7c57aa1715e5d"),
  "title" : "Jedi Lightsaber",
  "description" : "Conquer the galaxy or just look cool at your next cosplay conference",
  "unitPrice" : 99.25
}
```



# Document Retrieval

- Exact Match and “Starts With” Match:

```
// equality filter
db.productcollection.find({"title" : "Jedi Lightsaber"})

// starts with filter
var searchExpr = new RegExp(req.body.productTitle + "[a-zA-Z0-9]+", "i");
db.productcollection.find({"title" : searchExpr});
```

- Returning Specific Fields

```
db.productcollection.find({"title" : "Jedi Lightsaber"}, {title: 1, unitPrice: 1}).pretty()
{
  "_id" : ObjectId("53db8a2e85a7c57aa1715e5d"),
  "title" : "Jedi Lightsaber",
  "unitPrice" : 99.25
}
```

# Specifying Compound Filters

- “And” query

```
db.productcollection.find(  
  {  
    title : "Jedi Lightsaber",  
    unitPrice: {$gt: 50}  
  }  
)
```

- “Or” query

```
db.productcollection.find(  
  $or : [  
    {unitPrice: {$gt: 50}},  
    {title: "Jedi Lightsaber"}  
  ]  
)
```

# Accessing MongoDB From Node

- **mongodb** – A native mongodb driver.  
<http://mongodb.github.io/node-mongodb-native/>
- **monk**
  - Simplifies mongodb usage
  - Provides easier to use method signatures, built-in promises, auto-casting of `_id` in queries, and other features to make integration more developer friendly.
- Both modules are installable via npm

```
{  
  "name": "application-name",  
  "version": "0.0.1",  
  "private": true,  
  "scripts": {  
    "start": "node ./bin/www"  
  },  
  "dependencies": {  
    "express": "~4.0.0",  
    "static-favicon": "~1.0.0",  
    "morgan": "~1.0.0",  
    "cookie-parser": "~1.0.1",  
    "body-parser": "~1.0.0",  
    "debug": "~0.7.4",  
    "jade": "~1.3.0",  
    "mongodb": "*",  
    "monk": "*"   
  }  
}
```



# Connecting Mongo to Node

```
var express = require('express');
var path = require('path');
var favicon = require('static-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

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

var mongo = require('mongodb');
var monk = require('monk');
var db = monk('localhost:27017/NodeJSExamples');

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

app.use(favicon());
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use(function(req, res, next) {
    req.db = db;
    next();
});

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

# Accessing Mongo from Express Routes & Jade

```
router.get('/', function(req, res) {  
    var db = req.db;  
    var collection = db.get('productcollection');  
    collection.find({}, {}, function(e, docs) {  
        res.render('products', {  
            "products": docs  
        });  
    });  
});
```

extends layout

block content

```
h1= title  
table(border=1)  
  thead  
    tr  
      th Title  
      th Description  
      th Unit Price  
  tbody  
    - each item in products  
      tr  
        td= item.title  
        td= item.description  
        td(align="right") #{numeral(item.unitPrice).format('0.00')}
```



# Demo: Using MongoDB

- Install MongoDB
- Use NodeJS to save and retrieve Mongo documents

# Contact me!

- Steve Drucker
- e. [sdrucker@figleaf.com](mailto:sdrucker@figleaf.com)
- blog: <http://druckit.wordpress.com>
- facebook: [steve.drucker](https://www.facebook.com/steve.drucker)
- linkedin: [www.linkedin.com/in/uberfig](http://www.linkedin.com/in/uberfig)
- GitHub: [github.com/sdruckerfig](https://github.com/sdruckerfig)
- [www.figleaf.com](http://www.figleaf.com)
- [training.figleaf.com](http://training.figleaf.com)