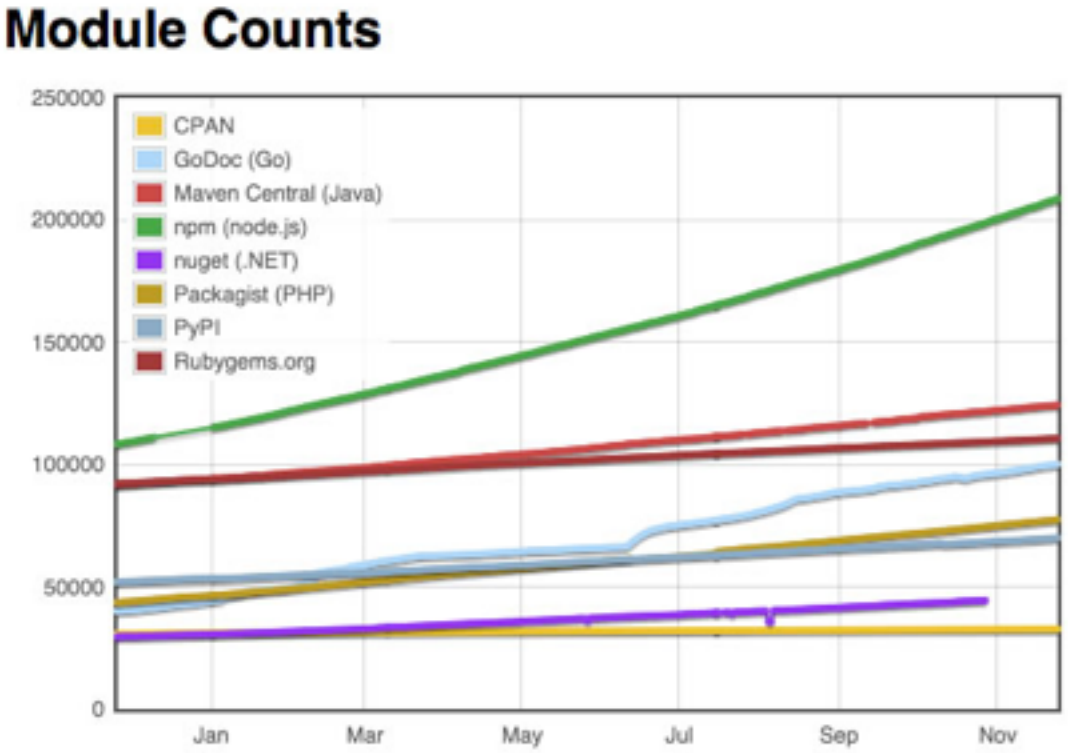
# PennApps Node.js Workshop

Justin Kim

# Why Node.js?

- Already using it for the browser
- Growing FAST



# Outline

1. Intro to Javascript
2. Intro to the Web
3. Node.js

# Installing Node.js

- Use Node Version Manager (NVM) to manage and install Node.js versions
  - Use this even if you plan on just using one version
- We will be using version 6.5.0
- When you have trouble remembering what methods do, use MDN

# Printing in Javascript

- You can print a value by passing a string to `console.log`
  - I will denote output with `// -->`

```
console.log('hello world'); // --> “hello world”
```

# Running Javascript

- Use a REPL (Read-Execute-Print-Loop):
  - with the `node` command in terminal
  - with the console in a browser
- Execute .js files with the `node` command:
  `node file.js`

# Literals

- Numbers: `1, 2, 3, 1.28e4, NaN, Infinity`
- Strings: `'xyz', 'foo \n', '\u2603'`
- Boolean: `true, false`
- Objects: `{ title: 'Javascript', language: true }`
- Array: `[1, 2, 'ham', 'spam']`
- Functions:

```
var square = function (x) {
  return x * x;
};
```

# Objects

- Lightweight, mutable key-value stores
- Literal notation uses curly braces
- Access with `obj.propertyName` or `obj['propertyName']`

```
var obj = {
  prop: 'hello'
}

obj.prop // --> "hello"
obj['prop'] // --> "hello"
```

# Functions

- First-class JS object
  - Allows JavaScript to use functional programming techniques
- Returns values with the `return` keyword
  - Otherwise, `undefined` is returned

```
var square = function (x) {
  return x * x;
};
```

# Functions vs Calls

- Don’t get confused with the difference between function calls and the function itself!
  - The call will always end with parentheses

```
var square = function (x) {
  return x * x;
};

console.log(square); // function
console.log(square(2)); // function call
```

# What is a callback?

- A callback is a function that's bound to a single asynchronous call
- It is passed as an argument to another function, with the expectation that it will be executed once some async task is finished

```
var cb = function () {
  console.log('callback ran!');
};
// wait 500ms, then run the callback
setTimeout(cb, 500);
// --> 'callback ran!'
```

# Node-Style Callbacks

- Since so many operations rely on callbacks, a standard callback has emerged in Node.js

```
var cb = function (err, results...) {...}
```

- `err` contains an error, if one occurred
  - Otherwise, it should be `null`
- After `err`, there can be any number of results arguments containing data

# Installing npm packages

- Node.js libraries are called packages
- The command to install them is `npm install package_name`
  - When installed, the package is installed in the current directory's `node_modules` directory
- To use a package, pass the name of the package as a string to the `require` function at the top of the file (e.g. `require('pry')`)

# package.json

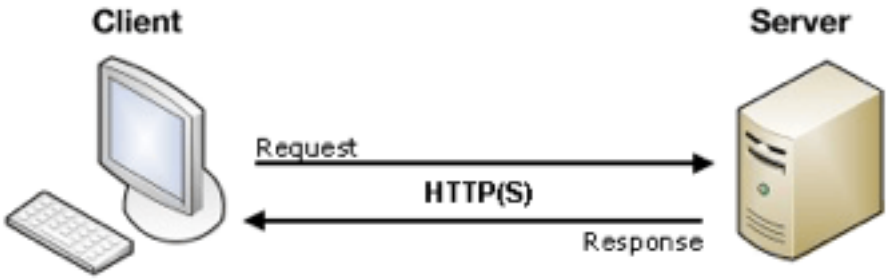
- Contains metadata for the app (e.g. name, author, etc.)
- Contains scripts
  - Can be called with `npm script` (e.g. `npm start`)
- Contains list of dependencies
  - Can be installed with `npm install`

```
{
  "name": "pennapps-nodejs-workshop",
  "private": true,
  "scripts": {
    "start": "node ./bin/run-server.js"
  },
  "author": "Justin Kim",
  "dependencies": {
    "body-parser": "1.8.1",
    "debug": "2.0.0",
    "ejs": "1.0.0",
    "express": "4.9.0"
  }
}
```

# The Web

# HTTP

- Stands for Hypertext Transfer Protocol
- A **client** (e.g. web browser, phone, computer, etc.) sends a **request** to a **server**
- The **server** receives this **request** and sends back a **response**
- This **response** is usually a web page (i.e. HTML with accompanying files) or data, usually in XML or JSON



# HTTP Verbs

- The five most common types of HTTP requests are:
  - GET
  - POST
  - PUT/PATCH
  - DELETE

# GET Request

- This is usually the default type of request sent
  - When you enter a URL or click a link, a GET request is sent for the web page
  - When a web page updates, it probably sent a GET request behind the scenes to get the new data
- It should only be used to *get* something

# POST Request

- This should be used to *send* data from the client to the server
- While you can technically use GET requests to send data as well, you should absolutely use POST requests if you're sending data
  - It's much more robust and secure
- This is the default type of request sent when submitting a form (e.g. log in)

# PUT/PATCH Request

- This should be used to *update* something on the server
- Technically, you can use a POST request to update as well, but it is convention to use a PUT or PATCH request
- The main difference between a PUT request and a PATCH request:
  - A PUT request is used to update an entire record
  - A PATCH request is only used to update part of it

# DELETE Request

- This should be used to *delete* something on the server
- Technically, you can use a POST request to delete as well, but it is convention to use a DELETE request

# Node.js

# What is Node.js?

- Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine
- Practically, this means you can now run JavaScript outside of the browser
- It is NOT a web framework

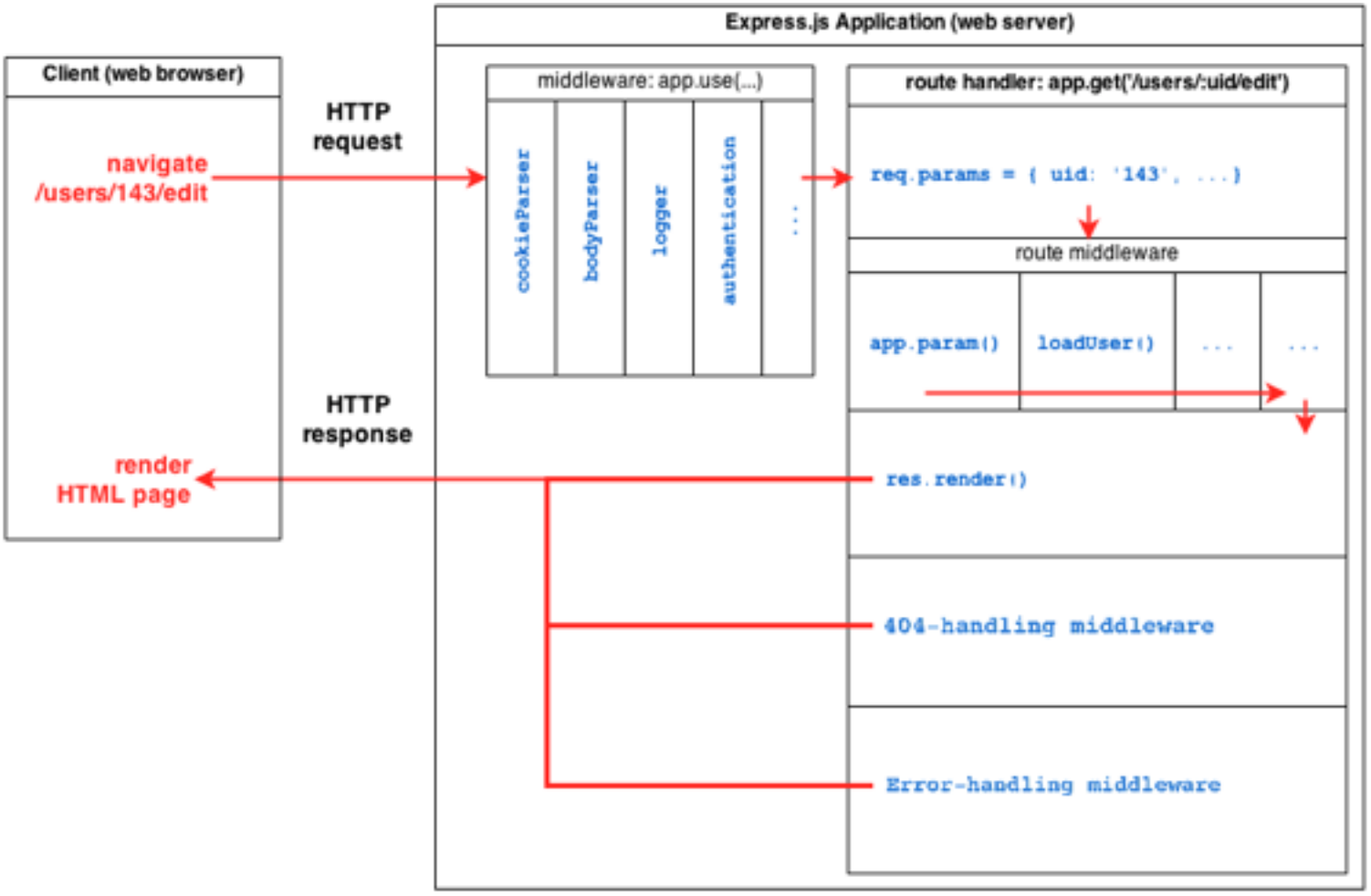
# Express

- Is a package – must install via npm
- Handles URL routing, requests, and responses
- Is oriented around middlewares and handler functions

# Hello World Express App

```
var express = require('express');
var app = express();
var port = process.env.PORT || 3000;
app.get('/', function (req, res) {
  return res.send('hello world!');
});
// Start listening for requests
app.listen(port, function () {
  console.log('Listening on port ' +
port);
});
```

# Anatomy of an Express App



# Middleware

- A **middleware function** is a function that handles a request
  - They are *chained together* so that multiple middlewares run on the same request
- Middleware parameters:
  - `req` - an object representing the request
  - `res` - an object representing the response. Has several methods (`.render`, `.send`, `.json`) to send data and complete the request
  - `next` - a callback (!) that passes control to the next middleware
- You can modify the `req` and `res` object directly - they are the actual objects that will be passed to other middlewares

```
var noOpMiddleware = function (req, res, next) {
    next();
};
```

# Adding Middleware in Express

- It's just a matter of using `app.use()`
  - Note, middlewares will run in the order you add them

```
app.use(function (req, res, next) {
    console.log('I am a middleware!');
    next();
});
```

# The Express Router

- Behaves like middleware
- Performs routing functions (i.e. handles requests to different urls)
- Used to modularize your code by defining subsections

# Router Example

```
var router = express.Router();
router.use(function (req, res, next) {
    console.log('I am a router!');
    next();
});
router.get('/', function (req, res, next) {
    res.send('Hello from the router!');
});
app.use('/router', router);
```

# Router Methods

- We've already seen that routers can:
  - use middleware
  - handle "get" requests
- You can actually handle ANY HTTP verb just by calling `router.verb()`
- Supported methods include:
  - `router.use()`
  - `router.get()`
  - `router.post()`
  - `router.put()`
  - `router.patch()`

# Route Parameters

- By putting a colon before a section of a route, you can create a *parametrized route*
  - with the parameter values available on `req.params`
- For example, if you have the route `/user/:id`, then it will match:
  - /user/1234 → req.params.id = '1234'
  - /user/justin → req.params.id = 'justin'
  - /user/id → req.params.id = 'id'
- Useful for creating a RESTful API

# Requests

- The request object is passed in to every middleware function in order
- Most of its properties are set by the middleware themselves
  - The `req.body` property is set by the body-parser middleware
  - The `req.cookies` property is set by the cookie-parser middleware

# Responses

- `res.set()` - sets a header value. Useful for allowing your app to be used from any site
  - `res.set('Access-Control-Allow-Origin', '*')`
- `res.status()` - set the HTTP status code to indicate an error
  - `res.status(404)` for Not Found errors
- `res.send()` - send a string, object, or Array as data
  - `res.send({error: 'Mocha exploded!'})`
- `res.redirect()` - redirect to another page
  - `res.redirect('/login')` - redirect to login page

# Local Variables

- To specify the values for variables in your template (i.e. the client), just modify `res.locals`
- This can be done with some middleware:

```
app.use(function (req, res, next) {
  res.locals.title = 'My Awesome Express App';
  next();
});
```

# Rendering

- Once you've set up your local variables, call `res.render(template)`
  - it'll render the template using the local values
- You can also pass in more local variables at call time

```
app.get('/', function (req, res) {
  res.render('index', {greeting: 'hi'});
});
```

# View Engine

- Express expects your templates to be in a `views` folder
  - you can use any (or many) libraries to process those templates
- Let's use EJS (Embedded JavaScript) to render our `.ejs` templates and also as the default

```
app.engine('ejs', require('ejs').renderFile);
app.set('view engine', 'ejs');
```

# EJS

- You can render JavaScript variables with `<%= variable %>`
- You can also run JavaScript directly on the browser with `<% CODE %>`
- These files typically have the extension `.html.ejs`

# Next Steps

- Look at MongoDB
  - Use mongoose as your ODM
- Look at jQuery
  - Useful for manipulating the DOM
- After all that, look at frontend frameworks
  - ReactJS
  - EmberJS
  - BackboneJS