

## Modules

# Simple web server reflection

---

1. What would your server do with a request with no path?
  - Would it return index.html? Why not?
2. What is the benefit of asynchronous reading the file and writing to the response?
  - What would be the difference if you used synchronous file reads?
  - What would happen to simultaneous visitors to your site?
  - What is a use case for using a synchronous file read?
3. There are two asynchronous file read options:  
    `fs.readFile` and `fs.createReadStream`
  - Look up both APIs
  - What is the tradeoff for this Simple Server task?
4. This assignment was to simply GET a file. How would you know if the request was a POST?
5. What network layers is this all working over. When you `response.write`, what protocols are being used?

# Modules – Separation of Concerns

---

- Modules allow for *separation of concerns*
  - I.e. separating your server program into files, each providing a distinct functionality.
- Three types of modules
  - Node Core API modules
    - Packaged in the basic Node installation
  - Contributed modules
    - Akin to the idea of RubyGems
    - Retrieved with Node Package Manager (NPM)
  - Local modules
    - I.e. the modules you develop
    - Your server should not be one monolithic file. It will be more understandable, maintainable, and parts reusable if functional parts are separated into individual files.

# Local Modules

---

- A module is a single JavaScript file
- Within the file, use *module.exports* for the functions and variables that you want to be accessible from outside the module
  - The scope of all variables and functions not exported will be restricted to within the file itself.
- *Require* modules to use them in another file  

```
var myDuck = require("./duck.js")
```

# Destructuring assignment

---

- See <https://nodejs.org/api/modules.html>
- Note in the example:  
`const { PI } = Math;`
- What is this?
  - Called a *destructuring assignment*
    - See [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\\_assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)
  - Math.PI is defined in the Math class
  - Therefore equivalent to:  
`const PI = Math.PI`

# module.exports

---

- Demo:
  - arithmetic
  - calculator
  - Point
  - calcserver

# exports vs module.exports

- exports is a variable equal to module.exports
- You can use exports to add to module.exports
  - exports.key = 123456
    - Equal to module.exports.key = 123456
  - exports.double = function(x) { return 2\*x};
- The module.exports property can be assigned a new value (such as a function or object).
  - module.exports = Point
- But assigning to exports will not modify module, must use module.exports
  - ~~exports = Point~~

# HW9 – Modules

---

## Task1: Create a module that exports methods & data

- Similar to calculator.js, create a module that has at least 3 methods and a variable
- Create a test program to demonstrate *require*-ing and using the module

## Task2: Create a module that exports a class

- Similar to point.js, create a class (you can reuse the one you created for HW1 – JavaScript Classes) module.
- Starting with your hw solution for today (HW8 – Simple Server)
  - Add a path that will take data from an http request
    - Use your class to do something
    - And then return the result to the client in the form of JSON
    - Don't break the default behavior of serving static files
  - Create an html form and use \$.getJSON to make an AJAX request to your server to use your class and display a result.
- Deploy to Now



# Submitting...

---

- Project organization:
  - "Task1" folder
    - Code files (module and test)
    - Module and test files should be clearly commented
    - No narrative necessary
  - "Task 2" document (only, no code)
    - URL of app deployed to Now
      - Be sure `/_src` is public if you have subscribed to Now
    - Narrative showing screenshots of
      - Form in browser
      - Form filled in and ready to submit
      - Result in browser
    - Module and server should be clearly commented
- Zip it all together and submit to Canvas

# Node Modules

---

- Node code is packaged into modules
- Modules are added to a program via require()
- You have seen this already in the Lab due today
  - `var http = require('http');`
    - The variable `http` becomes a handle to access the data and methods in the `http` module.
      - E.g. `http.createServer()`
    - Convention: The variable referring to a module is typically set to the same name as the module. But it is also legal to use another name.
      - E.g. `var birdhouse = require('http');`

# 3 Sources of Modules

---

- Node Core API modules
  - Packaged in the basic Node installation
- Contributed modules
  - Akin to the idea of RubyGems
  - Retrieved with Node Package Manager (NPM)
  - We will look at NPM on Tuesday
- Local modules
  - I.e. the modules you develop

# npm

---

- npm is the Node package manager
  - The author claims it is not an acronym
- It is installed alongside Node in the standard Node installation
- It is a command-line utility
  - npm help
- It is also available to browse and search:
  - <http://npmjs.org>
- Take some time to browse what is available
- As with any contributed software, research:
  - How recently it was updated
  - How often it has been downloaded
  - How many other modules depend on it
  - Have bugs been fixed recently

# npm Packages examples

---

- Microsoft Driver for Node.js for SQL Server
- Amazon S3 client
- Many frameworks for dealing with HTML & CSS
- Interact with Minecraft game servers
- Control DIRECTV boxes
- Control Parrot AR Drone quad-copters.



# package.json

---

- package.json
  - A file to set parameters for your app
    - e.g. Name, version...
  - And indicates the dependencies on other modules
- npm will use package.json to automagically download and install all modules you need and their dependencies
- package.json is also be used for additional directives when deploying your app to the cloud.
- See doc:
  - <https://docs.npmjs.com/files/package.json>

# package.json

- Example

```
{  
  "name": "application-name",  
  "version": "0.0.1",  
  "private": true,  
  "scripts": {  
    "start": "node app"  
  },  
  "dependencies": {  
    "express": "3.0.x",  
    "ejs": "*"   
  }  
}
```

How the node would be started,  
perhaps with parameters.  
The suffix .js is assumed

module dependencies  
used by npm  
3.0.0 would be a fixed version  
3.0.x is latest version within 3.0.  
"\*" is wildcard latest version  
It is best to constrain to versions  
you have tested.

# npm install

---

- Once you have defined package.json, then run  
npm install
- It will calculate all dependencies and download all modules
- Alternatively, to add a new module to package.json dependencies & download it:
  - npm install *package-name* --save



# Review: Sources of Modules

---

- Node core:
  - <https://nodejs.org/api/>
- Contributed @ NPM:
  - <https://www.npmjs.com>
  - Installed using the npm command line interface (cli)
- Part of your application
  - Separate JavaScript files
  - Using `modules.exports`
- In call cases, include in your Node.js program using:
  - `require()`

# Globals (no "require" necessary)

- global
- process
- require
- console
  - console.log(x)
    - Print x to the console
    - Can do formatting substitution
      - e.g. console.log("count %d", count)
      - %s – String
      - %d – Number (both integer and float)
      - %j – JSON
      - % - single percent sign (%)
- \_\_dirname *underscore underscore dirname*
  - The directory path of the current JavaScript file being executed
- \_\_filename
  - File name of the code being executed
- module
- exports
- SEE: <https://nodejs.org/api/globals.html>