Webservers

A program that

- accepts incoming web requests
- sends outgoing web response

Express

http://expressjs.com/

- A **framework** to help write webservers in NodeJS
- "framework"
    - Creates app IF code follows rules/structure
- One of many webserver frameworks
    - A commonly used one
    - Fairly "low-level"
        - Your code close to what sent to browser
- We use it to see what webservers do

Create a new package

1. Make a dir for your new project: `express-test/`
   - NOT in your repo (throwaway test)
2. Inside `express-test/` (`cd express-test`)
   - run `npm init -y`
   - Creates a new package (your package)
     - Using default answers to setup questions
     - Doesn't do anything remote
       - Only local changes
3. A `package.json` was created
   - take a look

Installing Dependencies

Our new package can track **dependencies**

- Still inside `express-test/`
- Run `npm install express`
  - Want to run in same directory as `package.json`
  - This is a **local install**
- Not with `sudo`, not as admin

Because we are "in" a **package**, `package.json` updates

- And a `node_modules/` was created

Is this true for NodeJS only?

- `package.json` is true for all Node-based projects
    - but is Node-specific (Not Java, Python, etc)
- Most langs have some **dependency management**
    - What libraries does this program need?
        - **Library**: Bundled functions
    - What version(s) of those libraries are needed?
    - What version is installed?
    - Ability to fetch and install/re-install
    - Ability to update

JSON

- JavaScript Object Notation
- **A text format listing structured data**
    - Easily translates to Javascript
    - Looks like JS
- Note it is TEXT, not JS
- Used by Node, Java, .NET, Python, etc (!)

JSON Can...

Can:

- Contain Numbers, Strings, Booleans, `undefined`
- Contain Objects
- Contain Arrays

"Contain" is a strange word here, as all JSON is text

- Not the actual data
- But JSON is text that can translate to JS

JSON Can Not...

Cannot:

- Have comments
- Store functions/methods
- Store construction/"class" information
- Store `null`
- Store Map() or Set() data

Simple and durable, but **highly limited** to data

JSON Formatting

More strict and limited than JS formatting:

- String quoting is **only double-quoting**
    - JS accepts single/double/backtick
- All **object keys must be quoted strings**
    - JS does not require quoted keys unless special characters
- **No trailing commas** in objects/arrays
    - JS (ES6+) allows trailing commas
- Whitespace still irrelevant

Example JSON

```
{
    "cat": {
        "name": "Jorts",
        "age": 3,
        "buttered": false,
        "toys": [ "mousie", "laser pointer" ]
    },
    "dogs": undefined
}
```

- Only allowed types
- Strings are double-quoted
- Object keys are double-quoted
- No trailing commas
- Looks like JS, but is actually text

JSON is used for more than JS!

- `package.json` is Node (JS?) specific
    - Other JS engines exist (bun, deno)
        - May use `package.json`
- **JSON** as a format is used with many languages
    - Like XML, YAML, CSV, etc
    - More on this later

`package.json`

A JSON file that every `npm`-using package has

- Contains information about the package
    - including dependencies
    - But also other info
        - version number, author, and more
- Use even for private packages
    - Even if only on your machine
        - But often on github/similar too

What does `package.json` give us?

- delete `node_modules/` and contents
- run `npm install` (in `express-test/`)
    - `npm install`, not `npm install express`
    - Recreates `node_modules/`
        - Reinstalls express
    - This is how we will test your work
    - This is how teammates stay in sync
    - This is how users use libraries
    - Often how the servers you **deploy** to work
    - Allows "delete `node_modules` and reinstall"
        - "turn it off and back on again" of node

`package.json` parts include...

https://docs.npmjs.com/files/package.json

- package `name`
- `version` (in **semver** notation)
- `dependencies` list
    - Lists version or minimum version
    - `devDependencies`
        - For those working on the package itself
- Author/repo info
- `license`
- `scripts`

`package.json` scripts

The `scripts` part of `package.json:

- Defines shell (command-line) commands
- `npm run SCRIPTNAME`
- e.g. script of `"greet": "echo Hello"`
  - prints "Hello" when you run `npm run greet`
- Used to collect/simplify commands for users
- Used to **build** your package (more later)

A few pre-defined script names don't require "run"

- e.g. `npm test` is the same as `npm run test`

Written well, scripts work on many operating systems

Versioning

There is no universal truth to version numbers

- May be marketing (MS Word)
- May be date-based (Minecraft betas)
- May be dev vs prod (Linux kernels)
- May be weird (TeX and MetaFont)

Hard to reliably parse, compare, understand

SemVer is an attempt at meaningful versions

- Not just JS, not just web, all software
- Even used for some non-software

Semver - Semantic Versioning

https://semver.org/

- X.Y.Z - three numbers
- ".x" means "any"
    - so 2.x means 2.(whatever)
- NOT like decimal
    - 1.1 is NOT 1.10
    - 1.10.0 after 1.9.x
        - 2.0.0 "later" than 1.9 and 1.10

Semver parts

MAJOR.MINOR.PATCH

- **Major version** is an API-breaking change
    - Or *likely* breaking change
- **Minor version** is a new feature
    - No breaking changes
- **Patch version** is a bug/security fix
    - No breaking changes
- 0.x.x means *nothing* is stable

Additional rules for betas and release candidates

package.json dependencies

Running `npm install` in a dir with a package.json

- npm will install all of the dependencies
  - (recursively) into `node_modules/`

`node_modules/` should NOT be put into version control

- `package-lock.json` normally SHOULD be
  - Not for this course though

When you `require()` a file without a path:

- npm looks in `node_modules/`

Dependencies Versions

What does the `dependencies` object mean?

- `x.y.z` = exact version
- `^x.y.z` = latest of this major version
- `~x.y.z` = latest of this minor version

Those versions are what `npm install` installs

- creates `package-lock.json`
    - records EXACT versions installed
    - USUALLY should be put in source control (git)
    - Used during deployment
        - recreate exact versions tested

Why so many dependencies

Is this SO MUCH CODE?

- Not really
- node libs tend to small and numerous
- Some other languages go for large and few

Dependencies (either style) ARE a security concern

- But often a necessary one
- Who to trust?
- How to learn about vulnerabilities?
- How to stay up-to-date on versions?

What is `package-lock.json`?

`npm install`

- creates a `package-lock.json` file `yarn` is an alternative to `npm`
- `yarn install` creates a `yarn.lock`

Both of these are **lockfiles**

- **lockfile** has many meanings
- we are talking Node only here

Node lockfiles

These **lockfiles** (`package-lock.json`, `yarn.lock`)

- Define exactly which dependency versions installed
    - `package.json` just defines options
    - And from WHERE
- During **deployment**
    - When server installs package for actual users
    - Don't want diff versions than deved/tested

Committing lockfiles to repo

On normal projects

- You should **commit your lockfiles to repo**
- Ensures exact matching versions can be installed
- Regularly updating dependency versions
    - Notable part of the job
    - Dependencies aren't "free"

For this course

- We ignore lockfiles
- Not managing dependency versions
- Using very limited libraries

Create your static assets

Inside `express-test`, create a `public` directory.

- This will hold **static** files and assets
- This will be the **document root** for **static assets**

Create an `index.html`

- inside `public/`
- that says "Hello World"

Basic Express Webserver

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

app.use(express.static('./public'));

app.listen(3000, () => {
  console.log('listening on http://localhost:3000');
});
```

Confirm the static assets

- run `node server.js` (from package directory)
- view `http://localhost:3000` in browser
    - Why 3000? Random but common dev port
    - Ports below 1024 require admin permissions
    - Common web dev ports:
        - 3000, 4000, 5000, 9000
        - 8000, 8080, 8443, 8888
    - Usually only 80/443 for real internet access
        - these dev ports only for local access
            - occassionally intranet/VPN access

Document Root and Static Assets

The `public/` directory is our **document root**

- Files in it are viewable using our server
- Links to these assets will NOT include `public/`
    - Common source of confusion

If we have:

```
public/
public/index.html
public/css/styles.css
```

- `<link rel="stylesheet" href="/css/styles.css">`
    - relative path also works
    - but no `public/` in `href` path

`express.static`

```
app.use(express.static(DOCUMENT_ROOT));
```

Ex: `app.use(express.static('./public'));`

- Defines directory to use as **document root**
- Will try to match paths and files from requests
  - To paths and files in that directory
- Often more complicated to
  - handle other Operating Systems
  - allow starting "from" different directories

```
const path = require('path'); // 'path' lib ships with Node
app.use(express.static(path.join(__dirname, 'public')));`
// __dirname is built-in special variable
```

Adding a dynamic asset

In `server.js` before `app.listen(...)`:

```
app.get('/dynamic.html', (request, response) => {
  response.send('This is not an actual file');
});
```

`public/index.html`:

```
<a href="dynamic.html">See a Dynamic page</a>
```

Restart `node server.js`

Confirm you can follow the link to the dynamic page

- A user can load `http://localhost:3000/dynamic.html`
- But there is no `dynamic.html` file

Why Restart?

Static asset changes **don't** require the server to restart

Dynamic asset changes **do** require the server to restart

Why?

Express Routes

We give Express a collection of **routes** and **callbacks**

If request matches the **route** (Method + Path)

- Call callback (the **handler**)
- Pass a **request** and **response** objects
- Callback will decide how to respond

Server loops through all routes for each request

- Stops at **first match**
- Each handler decides: stop or continue to next match

Summary - Webservers

Webservers are programs

- listen for incoming web request
- send a web response

Response may be a **static asset** (file)

- May be a **dynamic asset** (generated)

Response is always based on request

- Path may be used as file-system path
- Path may be used as data
- Path may be ignored!

Summary - Express

We run a server using `express` framework

- Our server **listens** to a **port**
- Each request passes through a series of checks
    - Does the request match these terms?
    - If so, call this **handler callback
        - Callback will decide to send response
        - ...and what to send
        - ...and if to pass on to more checks

This is a **Chain of Responsibility** pattern

Summary - Package.json

- `npm init` creates a **package.json**
- `package.json` has info on the package
- including the dependency list
- `npm install` will install the dependencies
    - needs to run inside the package directory!
    - same directory that has the `package.json` file
- `package-lock.json` normally committed to repos
    - `node_modules/` is NOT in repos

Summary - JSON

JSON is a **text format**

- Looks like JS, is not JS
- Easily translated to/from JS
- Only holds "data", nothing runnable
    - no functions
    - no classes
- Has strict formatting requirements
    - all quoting is double-quotes
    - object keys are quoted
    - no trailing commas
    - no comments