2 Sessions

Modules

Node Training

Modules

- Module Best Practices
- Advanced NPM Commands
- Third Party Modules

Rootless Global Install

```
$ mkdir ~/npm-global
$ npm set prefix ~/npm-global
$ echo "export PATH=$PATH:
  ~/npm-global/bin" >> ~/.profile
$ source ~/.profile
```

- Installing public modules with root permissions is somewhat risky
- Further, root permissions may not be available in a multi-user environment
- Creating a folder for global installs in the home directory preferred

Types

- Modules that could be published
- Application specific modules
  - Configuration modules
    - Such as routes in express, or tasks in grunt
- App utility modules
  - But could these actually be published?

NPM Initialisation Defaults

```
$ npm set init.author.name Joe
$ npm set init.license MIT
$ npm init
```

- Setting the author name avoids a monotonous step
- Altering the default license to your organizations main choice can really help to avoid accidental mis-licensing
- There's also `init.author.email` and `init.author.url`

Small and focused

- A module should be a single unit of functionality
- Do one thing, and do it well
- This is good for
  - Testing
  - Security
  - Third party vetting
  - Maintenance
  - Cognitive footprint

Use module.exports

```
(function (exports, require, module, __filename, __dirname) {
   //module code
});
```

- When Node loads a module the content of the code is wrapped in this function, prior to execution.
- `module = {exports: exports}`
- Export a single thing, instead of tagging multiple properties onto exports
- Assigning to exports would only override the param, so we assign to `module.exports`

State

- Never store global state
- Never store state at a modularly level
- Instead use instances to manage state

Expose a single function

```
function myMod() {
   //return an instance
}
myMod.sub = function sub() { }
module.exports = myMod;
```

- Functions are objects, they can have properties (and therefore, methods)
- Use the main function as an instance factory, or for the primary use case of the module
- Prefer factory functions over constructors (thus avoiding forgotten `new`, and global leakage)
  - The module could always use constructors internally, if need be

API's

- Avoid `new` for public API's
  - A module with greater than 10 public methods could need a rethink
    - Perhaps it should be split into smaller modules
- Depending on the use-case, splitting into smaller modules can mean publishing additional modules and using them as dependencies, or keeping modules local to the project
  - Case 1: Express 4
  - Case 2: Lo-dash Node

API's

```
//express 2:
app.use(express.bodyParser());

//express 3:
var bodyParser = require('body-parser');
app.use(bodyParser());
```

- Express 4 took core functionality from Express 3 and rolled it as a separate published NPM module
- This works for the Express framework because the middleware concept is highly composable and server configurations are highly customized on a case by case basis

API's

```
var _ = require('lodash-node');

//method categories:
var collections = require('lodash-node/modern/collections');

//individual methods:
var assign = require('lodash-node/modern/objects/assign');
```

- The `lodash-node` module runs parallel (and is auto-generated by) the lodash module
- There are many lodash methods, that are specific to the framework, in this case it makes more sense to split the methods up into independent modules local to the project
- This provides memory efficiency when only a small subset of functionality is required

API's

- Prefer to return native objects or primitives from methods, rather than constructed objects
  - Fractal API's can become mind twisters
  - If a plain object or array isn't good enough, is the returned data or functionality more complex that it needs to be
  - Perhaps with the exception of the main function, which could have good reasons to return a constructed object (e.g. an instance)
    - Constructed objects are easier to profile

Module Naming

- Especially in the case of utility modules, prefer long descriptive names
  - `line-separator-stream` instead of `streamliner`
  - Let the frameworks with the marketing budgets use the cool names, modules should be the obvious answer to a problem, they should be discoverable
- Standard convention is to separate names with dashes, rather than `under_scores` or `dot.seperators`.
- Lowercase is enforced, so camel-case etc. is out

Use Dependencies

- Don't write any code until it's been established that there are no modules that already solve the problem
- Even half-baked modules can be a starting point
- Compose modules out of problems that have already been solved
  - For common issues, the research time is far less than writing from scratch
  - Isolate the unusual or project-specific piece of a problem, locate the common pieces that have already been answered, use them as dependencies

Requiring

- All `require`'s should go at the top
  - If all dependencies can be ascertained with a `head`command then the module has clear dependencies
  - Whilst it does look super awesome, try to avoid inlining requires
    - `app.use(require('body-parser')())`

Requiring

- Group require calls into sets:
  - Core modules
  - Public modules
  - Local modules

Versioning

```
$ npm version
```

- Ensure that only bug fixes with no breaking changes bump the patch number
- Compatible API changes should bump the minor version, but it is crucial that these changes are verified compatible
  - If in doubt, bump the major version
- Some modules never bump the patch version, only minor or major, this can help to avoid dependency breakage (less so with the npm 2.0)

Dev Dependencies

```
$ npm i mocha --save-dev
$ npm i --production
```

- Separate development dependencies from production dependencies
  - Development dependencies tend to be tools - test harnesses, linters, transpilers, etc.
- Use the `--production` flag, or set `NODE_ENV` environment variable to production to avoid installation of development dependencies

Peer Dependencies

```
{
  "name": "chai-as-promised",
  "peerDependencies": {
    "chai": "1.x"
  }
}
```

- In the `package.json` file, the `peerDependencies` field is used to specify that a module works *alongside* another module, and depends on that module being present.
- This is used for the likes of plugin modules
- A peer dependency will be installed alongside the plugin module if it isn't already present

Extraneous Dependencies

```
$ npm install chinese-random-skill
$ npm ls
$ npm prune
```

- An extraneous dependency is one that exists in the `node_modules` folder but isn't declared in `package.json`
- Use the `--save` flag to include the dependency in package.json upon install
- Use `npm ls` command to discover an extraneous dependencies before publishing

Optional Dependencies

```
$ npm i underscore lodash --save-opt
```

- An `optionalDependencies` field can occur in the `package.json` file
- If dependencies in this field fail to install the module install will still be considered successful
- This could be useful for including C++ modules with JavaScript fallbacks in case of build failure
- It could also be used for cross-realm modules (e.g. optional private hosted dependencies)

Loose Dependencies

- When performing an `npm --save` or any variant, npm will add the dependency to `package.json` using *loose versioning*
- This means when a module is distributed, the module consumer may receive different dependencies to those installed on an engineers system

Loose Dependencies

- In earlier versions of npm, the version would be prefixed with a tilde, e.g. `~1.2.3` which means treat the patch number as loose (i.e `1.2.x`)
- In the latest npm (bundled with Node versions above 0.10.16), versions are prefixed with a caret, e.g. `^1.2.3` meaning the minor version is also loose (i.e. `1.x.x`)

Tightening Dependencies

```
$ npm set save-prefix '~' #patch
$ echo "save-prefix=''" > .npmrc
```

- The `save-prefix` configuration option can be altered to return npm to patch-gradient looseness
- For critical modules, we may wish to ensure `package.json` versions are locked upon save
  - An `.npmrc` in the module folder can contain any configuration option and apply it locally to that module
- During development however, it's recommended to at least update to latest patch numbers to obtain bug fixes

Outdated Dependencies

```
$ npm outdated
$ npm update
```

- The `outdated` command will find and display any currently installed modules are out of date
- This is an important step prior to publishing, to help ensure module consumers receive the same dependencies as currently installed and tested against
- The `update` command can then be used to update the modules automatically, or else dependencies can be locked in by editing `package.json` or using `shrinkwrap`

Shrinkwrap

```
$ npm shrinkwrap
```

- The `shrinkwrap` command locks all dependency versions to a depth of infinity
- It generates an `npm-shrinkwrap.json` file thats essentially a fractal `package.json` file with fixed dependencies based on versions currently installed
- The `npm install` command will honour the `npm-shrinkwrap.json` file above the `package.json` file

Entry Point

- The `package.json` file has a `main` field, if unset the main module file defaults to `index.js`
- This is also true for requiring directories, for instance in express the routes folder has an index.js folder. Simply `require('./routes')` to get all routes
- Sticking to this convention instead of using custom entry point file names makes it easier for other people to get started with a codebase

Tests

- Prioritize tests for infrastructural modules
  - Utility libraries
  - Core functionality
- The `npat` option will run tests when installing dependencies, if tests do not pass the install process fails

```
npm install some-module --npat
```

Ignoring

- Ignore everything a module consumer doesn't need to achieve fastest install time
  - Ignore example code
  - Ignore generated docs
  - Ignore tests (maybe..)

Ignoring

- A blacklist or whitelist approach can be taken
  - Blacklist using `.npmignore` (works exactly like `.gitignore`)
  - Whitelist using `package.json files` array
  - Whitelisting often leads to tighter packages

Publishing

```
$ npm publish
```

- Running npm publish in a module folder will publish the module
  - As long as the version number has been altered
- Publishing over old versions is prohibited, even a change to the readme requires a new version

Unpublishing

```
$ npm unpublish --force mod@x.x.x
```

- Unpublishing is generally not recommended
- But definitely handy if, for instance, an engineer accidentally published code that contains AMI keys
- To unpublish the `--force` flag, module name and version number must be specified.
- The unpublished version number can no longer be used, a new version number has to be assigned

Tagging

```
$ npm publish --tag 4.0.0-rc.1
$ npm tag myMod@1.19.22 classic
```

- Tags are useful for publishing pre-releases, allowing general users to install the latest stable version where beta testers can install the release candidate according to its tag (e.g. `npm i myMod@4.0.0-rc.1`)
- They can also be used to accompany continued support for a prior version of a module (e.g. a version that could be suited to lighter use cases)

```
$ npm install myMod@classic
```

Environments

- Code should aim to work both server and client side where possible
  - Break out generic logic into its own module so that it can be reused
- Where applicable (e.g. utilities), modules should have both programmatic and command-line interfaces
- It may not be clear at first why this is necessary, but others could find excellent use cases
  - even coding with a view to multi-environment usage without actually implementing can help

Environments - Browser

```
{
  "name": "myMod",
  "browser": {
    "./lib/dep.js": "./lib/browser-dep.js",
    "public-mod": false
  }
}
```

- The `browser` property of `package.json` can be used to shim or otherwise disable both local and public modules prior to building a module for client-side use.
- The module can be built either directly or whilst a dependency using browserify

Environments - Browser

```
$ npm i -g browserify
$ browserify . > myMod.js
$ browserify . -s myMod > myMod-umd.js
```

- The `browserify` command line tool will prepare a Node module for browser usage
  - It polyfills many of Node core API's
- The `-s` option means standalone, this generates a UMD module which is compatible with AMD, CommonJS module loaders, falling back on to global export

Environments - CLI

```
{
  "name": "myMod",
  "bin": {
    "myMod": "./cmd.js"
  }
}
```

- The `bin` property of `package.json` can be used to specify
  - the desired name for a globally installed executable (the object key)
    - should generally be the same name as the module
- the executable file which the name will be linked to (the object value)

Environments - CLI

Creating a command line app

```
#!/usr/bin/env node

var myMod = require('./index.js')
var fs = require('fs')
var path = require('path')
var argv = require('minimist')(process.argv.slice(2));

if (argv.help) {
  return console.log(
    fs.readFileSync(path.join(__dirname,
      'usage.txt')) + ''
  )
}

//do the rest
```

Ownership

```
$ npm owner add newowner
$ npm owner rm `npm whoami`
$ npm owner ls
```

- Transferring ownership is a two step process
  - Add the new owner
  - Remove yourself as owner
- Of course, multiple owners can be added for team managed modules

Local Modules

```
{
  "name": "myMod",
  "private": true
}
```

- Modules that aren't intended for public consumption can be labelled private in cases of unintentionally calls to `npm publish`
- This could potentially be used in conjunction with private github repos, which are referenced instead of version numbers
  - This is not a great approach

Self Hosted Modules

```
{
  "name": "orgsMod",
  "publishConfig": {
    "registry": "http://onsite.npm-reg.local/"
  }
}
```

- The `publishConfig.registry` option can point to a self-hosted NPM registry
- Historically setting up a local registry has been far from trivial

Scoping

```
{
  "name": "@myScope/myMod"
}
```

- npm 2.0+ supports scopes
- npm 2.0 comes with io.js and Node 0.12, but can be installed with 0.10 separately
- npmjs.com has supported scoped packages since early 2015

nearForm

Scoping

```
{
  "name": "@myScope/myMod"
}
```

- Scoped packages can be publicly or privately published
  - Private publishing requires paid account
  - Unlike GitHub, every collaborator must also have a paid account

Scoped Registries

```
npm set @myScope:registry http://onsit
e.npm-reg.local/
```

- Scopes can be associated with self-hosted registries
- Until npmjs.com has Organizational accounts, this still remains the best option
  - And may still after that, depending on industry and company priorities

Locally Hosted Private Modules

```
$ npm i -g sinopia
$ sinopia &
$ npm adduser --registry http://localhost:4873/
```

- There is a module that makes locally hosted private modules extremely easy
- Sinopia spins up a server that talks to the npm client
- When installing dependencies, Sinopia will install from public npm when the module isn't contained in Sinopia's registry
  - Modules are also cached

Locally Hosted Private Modules

```
{
  "name": "@myOrg/myMod"
  "publishConfig": {
    "registry": "http://localhost:4873/"
  }
}
```

- For clarity, to avoid accidental public module overriding, and to be future ready locally hosted modules should be namespaced using the upcoming *scoping* syntax.
- In a rollout, sinopia would be deployed within an organisations network - there's a docker image to assist with deployment
  - Sinopia's storage directory is configurable, maybe using a shared network drive folder would work

Upgrading Node

```
$ npm rebuild [folder...]
```

- If C++ modules are being used, these will probably need to be rebuilt with the new node binary
- The `npm build` command rebuilds any C++ dependencies in a module
- `npm rebuild` calls npm build in the list of module folders specified

Evaluating Modules

- Use community resources to find the module you need:
  - http://npmjs.org search
  - http://nodezoo.com
- Review the http://npmjs.org package page for the module
  - How many dependants
  - Who wrote it? Are they active?

Evaluating Modules

- Review the Github page for the module
  - number of stars, forks, issues
    - However, lot's of interest can signify a scope that's too broad (violating do one thing well)
  - how recent is the last update
    - This is not always important, sometimes a module is feature complete.
  - Does the module have tests written for it?
    - Are they good tests?

Evaluating Modules

- Has coverage been calculated?
- How extensive are the docs?
- What about examples?
  - Do they look straightforward?

Production-level Modules

require	author	description
request	mikeal	HTTP client
hyperquest	substack	Lighter HTTP client
needle	tomas	Lighter HTTP client
minimist	substack	Command line options
async	caolan	Async patterns
mocha	tjholowaychuk	Unit testing

Production-level Modules

require	author	description
tape	substack	Unit testing
chai	jakeluer	Assertions
underscore	jashkenas	Functional patterns
lodash	jdalton	Faster functional patterns
nconf	indexzero	Configuration
passport	jaredhanson	Login and authentication

Production-level Modules

require	author	description
browserify	substack	Browser distribution
event-strm	dominictarr	Stream utilities
JSONStream	dominictarr	Stream utilities
through2	rvagg	Stream utilities
split2	matteo.collina	Stream utilities
pump	mafintosh	Stream utilities

Production-level Modules

require	author	description
duplexify	mafintosh	Stream utilities
xml2js	leonidas	XML to JavaScript
bunyan	trentm	Logging
express	tjholowaychuk	Server framework
hapi	hueniverse	Server framework
restify	mcavage	REST API builder

Production-level Modules

require	author	description
moment	timrwood	Date manipulation
debug	tjholowaychuk	Debug printer
ws	einaros	Websockets
socket.io	rauchg	Realtime
levelup	rvagg	LevelDB
mongodb	christkv	MongoDB

Production-level Modules

require	author	description
redis	mjr	Redis
pg	brianc	Postgres
mysql	felixge	MySQL
gulp	contra	Build system
grunt	cowboy	Build system

Production-level Modules

require	author	description
minimatch	isaacs	Glob matching
glob	isaacs	Glob matching
bl	rvagg	Binary parsing
dockerode	apocas	Docker management

Where Next?

- https://docs.npmjs.com
- http://substack.net/finding_modules
- http://nodezoo.com