Node.js Module System

By the end of this section, you should be able to:

- Generate a package.json
- Install a package from npm
- Understand the difference between production/development dependencies
- Understand semantic versioning
- Understand the differences between CommonJS and ESM
- Do *all* the importing

npm command

The npm executable is made available when you install Node.

To check it's working, in your terminal run

```
npm -v
```

You can see all of the available commands with

```
npm --help
```

and get help on a particular command with the help flag

```
npm install -h
```

Initializing a Package/App/Service/Node.js Project

A package is a folder with a package.json file in it (and hopefully some code).

We create a new directory and run

```
npm init
```

For the wizard that comes up, you can accept the default answers given in brackets by pressing enter.

This could be achieved without the wizard by running

```
npm init -y
```

Install dependencies

Now we have a package.json, we can install dependencies.

Let’s install a logger:

```
npm install pino
```

After the installation three things have happened:

- the package.json dependencies object has been updated
- there is now a package-lock.json file
- there is now a node_modules directory

```
node -e "require('pino')().info('testing')"
npm ls
```

Development dependencies

Not all dependencies are required for production, some are tools to support the development process.

These types of dependencies are called development dependencies.

To save something in our package as a development dependency we use the `--save-dev` flag.

```
npm install --save-dev <package_name>
```

If we only want to install production dependencies, we can do this with the `--production` flag. This will ignore all of the devDependencies.

```
npm install --production
```

Semantic versioning (semver)

Each of our dependencies have a semver range. Our package has a semver version.

```
{
  "version": "0.0.1",
  "dependencies": {
    "pino": "^7.8.1"
  },
  "devDependencies": {
    "standard": "^16.0.4"
  }
}
```

The format is MAJOR.MINOR.PATCH.

We can install a package at particular semver using the `@` symbol:

```
npm install pino@7.8.1
```

We can use `x` to give a more generic semver. This will install major 7, minor 8 and the latest patch.

```
npm install pino@7.8.x
```

By default, package installations have a caret(^) before the semver. This translates to MAJOR.x.x.

```
"standard": "^16.0.4"
```

Package scripts

The "scripts" field in package.json can be used to define aliases for shell commands that are relevant to a Node.js project.

We add scripts here to help with linting, development or building.

Let's add a linting script to our project:

```
  "scripts": {
    "test": "echo "Error: no test specified" && exit 1",
    "lint": "standard"
  },
```

To run this, we use `npm run`:

```
npm run lint
```

CommonJS vs ESM

When Node was created, JS didn't have a module system so they used require.js.

This approach is still used in lots of packages and documentation.

ESM is the approach that browsers use and Node has allowed for it since Node 10. More and more authors are moving to this approach.

Let's look at how we can export/import using CommonJS and then refactor to ESM.

All the importing

Exercises

1. Open `Labs/05-node-js-module-system/add.js` and create a function that takes two numbers and adds them together. Export that function.
2. In `index.js` import the function you've created and use it to console.log two numbers being added together.
3. Add a console.log to your `add.js` file which will add two numbers together loaded as a separate program. Check that this works by running `node add.js` in the directory.