Fundamentals of Web Applications
COMP 2406A – Winter 2020

NPM: Node Package Manager

Dave McKenney
david.mckenney@carleton.ca

Learning Outcomes

by the End of this Lecture, Students that have Completed the Reading Assignment and Review Questions should be Able to:

Understand how Node.js finds external modules

Use NPM to install external modules

Use NPM to manage dependencies

Use NPM to manage building of your projects

Node Modules

We have now seen how to specify our own modules in local files, as well as how to require built-in Node.js modules

Now we will see how external modules work

The Require Statement

When you require a non-relative path, like: require(“some_module”);

Node.js first looks for a built-in module

If it doesn’t find it, it looks inside the local “node_modules” folder

The Require Statement

When you require a non-relative path, like: require(“some_module”);

If “some_module” is not in the “node_modules” folder in the local directory, the search continues up the directory tree

The Require Statement

For example, if server.js is in this directory:

/Users/dave/COMP2406/myserver/

And server.js includes `require("express");`

/Users/dave/COMP2406/myserver/node_modules is checked first

The Require Statement

For example, if server.js is in this directory:

/Users/dave/COMP2406/myserver/

And server.js includes require("express");

/Users/dave/COMP2406/node_modules is checked second

The Require Statement

For example, if server.js is in this directory:

/Users/dave/COMP2406/myserver/

And server.js includes require("express");

/Users/dave/node_modules
is checked third

The Require Statement

For example, if server.js is in this directory:

/Users/dave/COMP2406/myserver/

And server.js includes require(“express”);

/Users/node_modules
is checked fourth

The Require Statement

For example, if server.js is in this directory:

/Users/dave/COMP2406/myserver/

And server.js includes require(“express”);

/node_modules
is checked last

The Require Statement

When the module is found, the checking stops (obviously?)

This allows you to manage the versions of modules used by your modules, to a degree…

Directory Structure and Require

If you have multiple projects that require the same module, you can organize your directory structure like:

```
my_project_folder
        |-- node_modules/my_module.js
        |-- my_first_project/server.js
        |-- my_second_project/server.js
```

Require from both server.js files finds the same my_module.js file

Directory Structure and Require

If you need a newer version of a module for one specific project:

```
my_project_folder
        |-- node_modules/my_module_v1.js
        |-- my_first_project/server.js
        |-- my_second_project/server.js
        |-- my_third_project
                |-- node_modules/my_module_v2.js
                |-- newserver.js
```

newserver.js finds my_module_v2.js

Even More Modules

The previous method works, but will get confusing for larger projects with many versions

Additionally, we have to build the directory structure ourselves (find modules, ‘install’ them, etc.)

Node Package Manager

We will use NPM (Node Package Manager) throughout the course to manage our project and their dependencies

It will allow us to easily add external modules to our projects and easily re-build our projects in other locations

Node Package Manager

NPM is a combination of:

The NPM website - https://www.npmjs.com

The NPM command-line tool

The NPM package registry

Node Package Manager

The package registry is essentially a giant repository of existing packages

A package is just a directory with 1+ modules, along with a package.json file

These are largely Node-based, but there are also front-end packages

You can use NPM to:

1. Incorporate packages into your projects
2. Share your own code with others easily
3. Create organizations and teams
4. Manage multiple versions/dependencies
5. Update projects easy when required code changes
6. Find other developers working on similar projects
7. Download standalone tools to assist in developing

Node Package Manager

NPM makes it easy to have many projects with varying dependencies

Projects can all require different modules and/or different versions of those modules

And NPM handles all of this for us, instead of requiring us to build up a complex directory system (there is file duplication, but not a big deal)

The package.json File

The package.json file is one of the most important elements of a project/package

It contains metadata about the project, such as project name, author, versioning, dependencies, etc.

The package.json File

An example package.json file:

```
{
  "name": "testpackage",
  "version": "0.0.1",
  "description": "Not really anything.",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1“
  },
  "author": "dave",
  "license": "ISC“
}
```

The package.json File

Along with basic metadata, the package.json file contains information about the package's dependencies

Allows you to specify specific versions of required packages

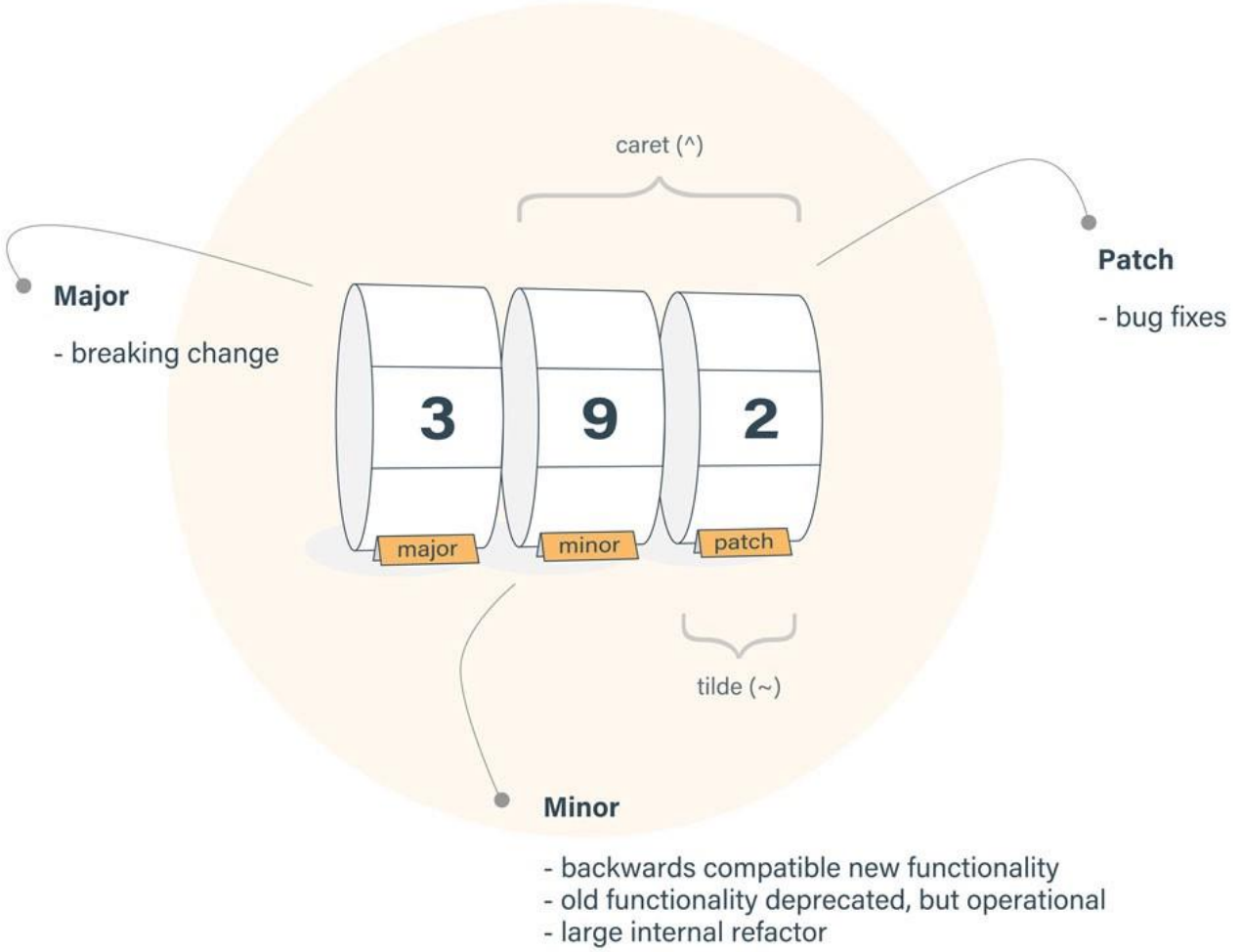
Allows you to automatically build the package (later)

The package.json File

```
{
 "name": "testpackage",
 "version": "0.0.1",
 "description": "Not really anything.",
 "main": "index.js",
 "scripts": {
  "test": "echo \"Error: no test specified\" && exit 1“
 },
 "author": "dave",
 "license": "ISC",
 "dependencies": {
  "express": "^4.17.1",
  "request": “~2.88.0“
 }
}
```

Within dependencies, key is the package name and value is semantic versioning information

Semantic Versioning



https://medium.com/beginners-guide-to-mobile-web-development/introduction-to-npm-and-basic-npm-commands-18aa16f69f6b

The package.json File

```
{
 "name": "testpackage",
 "version": "0.0.1",
 "description": "Not really anything.",
 "main": "index.js",
 "scripts": {
  "test": "echo \"Error: no test specified\" && exit 1“
 },
 "author": "dave",
 "license": "ISC",
 "dependencies": {
  "express": "^4.17.1",
  "request": “~2.88.0“
 }
}
```

Within dependencies:

^ indicates newer minor release or patch version

~ indicates newer patch version only

Creating a package.json File

To create a package.json file for your own package, navigate to its directory in command line and run:

npm init

This will ask you some questions about the package and create the base package.json file

Installing External Modules

To install an external module in a directory, run:

npm install *PackageName*

This will install *PackageName* into the node_modules directory, including any of its dependencies, and update the package.json file

Installing External Modules

For example, a module we will use in the future:

npm install express

You can also specify version information:
npm install *PackageName*@1.03
npm install *PackageName* @“~1.03”
npm install *PackageName* @“^1.03”

Removing External Modules

To remove a package that you have installed:

npm uninstall *PackageName*

or

npm uninstall --save *PackageName*

Updates External Modules

And to update dependencies:

npm update

This updates based on semantic versioning information from package.json

Installing from a package.json File

Finally, if you have a package.json file within the directory you are currently at:

npm install

This will install all dependencies listed in the package.json file, giving you an easy way to give somebody else your project

Installing from a package.json File

To ensure the same versions are installed, you can use the package-lock.json file

This was designed to include specific versions

Installing Global Packages

There is also an option to allow you to install packages globally:

npm install –g *SomePackage*

This is generally used only for command-line tools

Questions

Now that we can install external modules, we can use some of them to make our lives easier (or harder?)

Questions?