COM644 Full-Stack Web and App Development

Practical A2: npm – The Node Package Manager

Aims

- To introduce the Node Package Manager npm
- To examine the role of **package.json**
- To demonstrate the addition of new packages through npm
- To demonstrate how **package.json** can re-build a Node application
- To understand the **SemVer** Semantic Versioning Scheme for software
- To examine the role of scripts in **package.json**
- To specify a new script and test it

Contents

A2.1 Introducing npm

npm (Node Package Manager) is a Node tool that allows us to define and manage dependencies between code modules, libraries and packages. This helps promote the notion of re-usable code, where members of the Node community develop and share useful packages that others can include in their projects.

All npm packages contain a file, usually in the project root, called **package.json**. This file holds various metadata relevant to the project and is used to give information to npm that allows it to identify the project as well as handle the project's dependencies. It can also contain other metadata such as a project description, the version of the project in a particular distribution, license information, even configuration data - all of which can be vital to both npm and to the end users of the package. The **package.json** file is normally located at the root directory of a Node.js project.

A2.1.1 Creating package.json

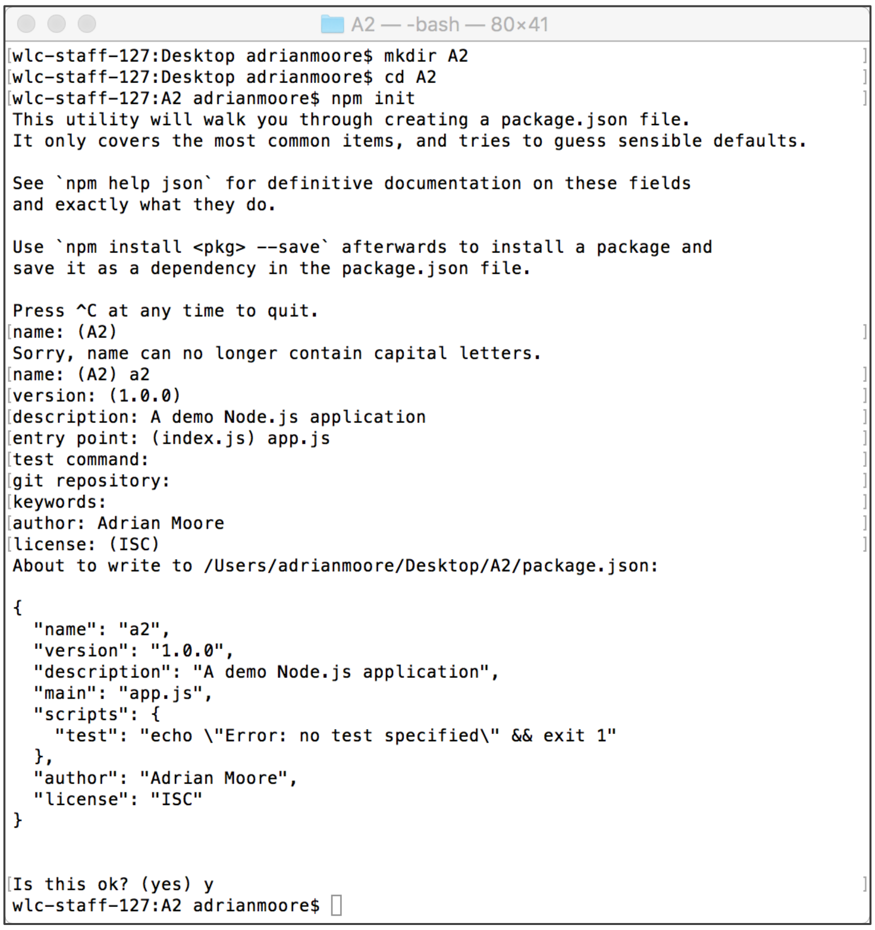
`The easiest way to create a **package.json** file is through the Node command line interface. Create a new folder called **A2** in which to manage all of the files generated in this Practical and navigate into it.

The new node project is then created by issuing the command

```
U:\A2> npm init
```

This launches a series of prompts that invite you to give context information about your project. We do not need to provide information for all prompts, but the information that we do provide is written in JSON format to the root node of our project.

Answer the prompts using the information in Figure A2.1 as a guide, and verify that the file **package.json** with the following content is created in response. (***Note:*** *Press <return> to accept the default answer for any prompt.*)



```
wlc-staff-127:Desktop adrianmoore$ mkdir A2
wlc-staff-127:Desktop adrianmoore$ cd A2
wlc-staff-127:A2 adrianmoore$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (A2)
Sorry, name can no longer contain capital letters.
name: (A2) a2
version: (1.0.0)
description: A demo Node.js application
entry point: (index.js) app.js
test command:
git repository:
keywords:
author: Adrian Moore
license: (ISC)
About to write to /Users/adrianmoore/Desktop/A2/package.json:

{
  "name": "a2",
  "version": "1.0.0",
  "description": "A demo Node.js application",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Adrian Moore",
  "license": "ISC"
}


Is this ok? (yes) y
wlc-staff-127:A2 adrianmoore$
```

Figure A2.1 Creating package.json

File: *A2/package.json*

```
{
  "name": "a2",
  "version": "1.0.0",
  "description": "A demo Node.js application",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Adrian Moore",
  "license": "ISC"
}
```

A2.1.2 Adding additional packages

The **package.json** file created for us is only the starting point of our application. There is a vast range of Node packages and libraries that we might also want to include and we can illustrate this by adding a dependency to **Express** to our application. (Express is a Node.js library that supports the creation of web server applications – we will investigate it further in Practicals A3-A6).

For now, we will install Express to our application by the command

```
U:\A2> npm install express --save
```

Note: The **--save** flag on the command instructs Node to install the package AND ALSO update package.json accordingly. Without --save, the library would be added to the application, but package.json would not be updated

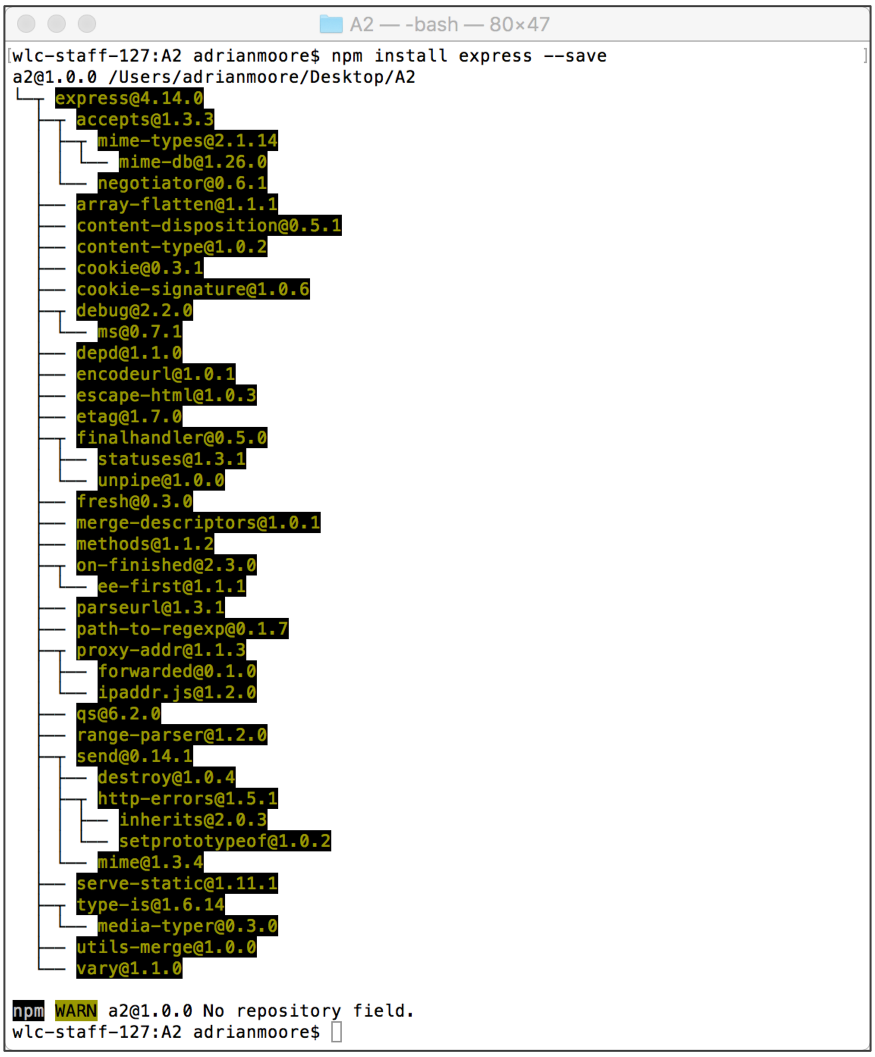
Add the **Express** package to your application and check that you get output such as that shown in Figure A2.2.

Your **package.json** file should also have now updated to include a new **dependencies** section specifying the version of Express to be included.

File: ***A2/package.json***

```
{
...

  "dependencies": {
    "express": "^4.16.6"
  }
}
```



```
wlc-staff-127:A2 adrianmoore$ npm install express --save
a2@1.0.0 /Users/adrianmoore/Desktop/A2
└─┬ express@4.14.0
  ├─┬ accepts@1.3.3
  │ ├─┬ mime-types@2.1.14
  │ │ └── mime-db@1.26.0
  │ └── negotiator@0.6.1
  ├── array-flatten@1.1.1
  ├── content-disposition@0.5.1
  ├── content-type@1.0.2
  ├── cookie@0.3.1
  ├── cookie-signature@1.0.6
  ├─┬ debug@2.2.0
  │ └── ms@0.7.1
  ├── depd@1.1.0
  ├── encodeurl@1.0.1
  ├── escape-html@1.0.3
  ├── etag@1.7.0
  ├─┬ finalhandler@0.5.0
  │ ├── statuses@1.3.1
  │ └── unpipe@1.0.0
  ├── fresh@0.3.0
  ├── merge-descriptors@1.0.1
  ├── methods@1.1.2
  ├─┬ on-finished@2.3.0
  │ └── ee-first@1.1.1
  ├── parseurl@1.3.1
  ├── path-to-regexp@0.1.7
  ├─┬ proxy-addr@1.1.3
  │ ├── forwarded@0.1.0
  │ └── ipaddr.js@1.2.0
  ├── qs@6.2.0
  ├── range-parser@1.2.0
  ├─┬ send@0.14.1
  │ ├── destroy@1.0.4
  │ ├─┬ http-errors@1.5.1
  │ │ ├── inherits@2.0.3
  │ │ └── setprototypeof@1.0.2
  │ └── mime@1.3.4
  ├── serve-static@1.11.1
  ├─┬ type-is@1.6.14
  │ └── media-typer@0.3.0
  ├── utils-merge@1.0.0
  └── vary@1.1.0

npm WARN a2@1.0.0 No repository field.
wlc-staff-127:A2 adrianmoore$
```

Figure A2.2 Install Express

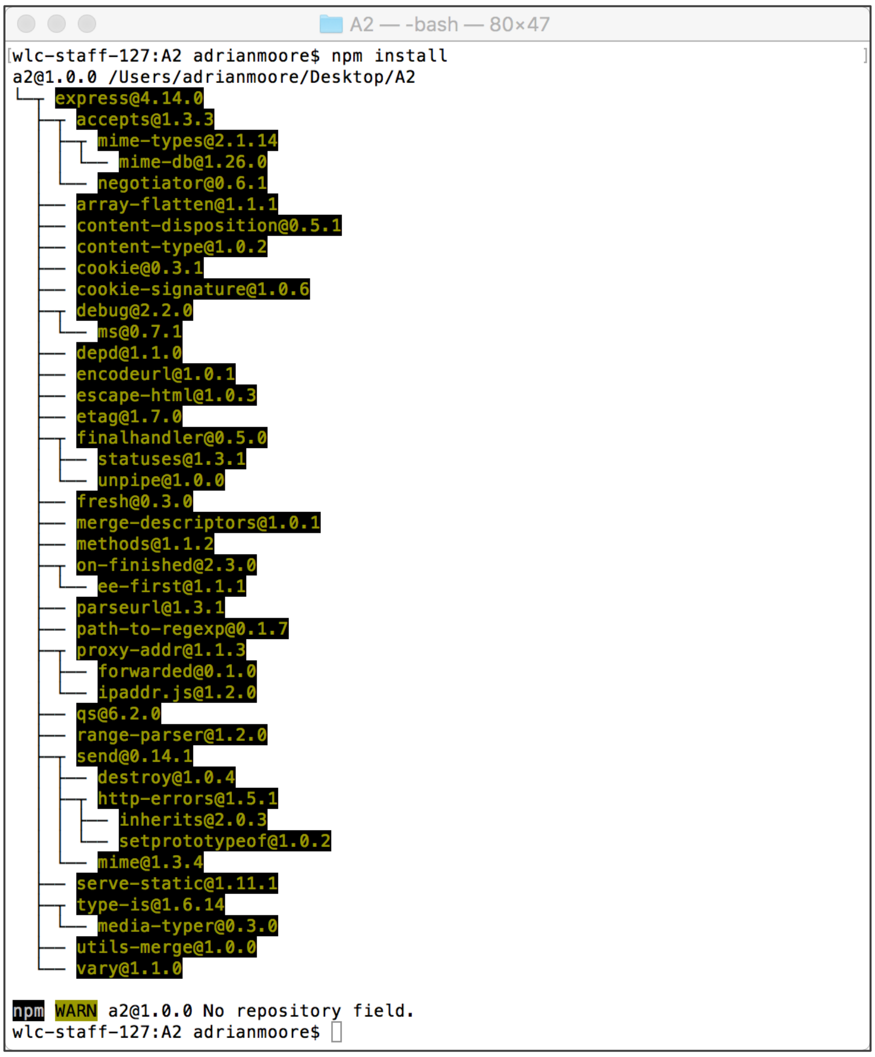
The **node_modules** folder is where external code packages are managed within our application, but if we are sharing or archiving our code, we do not want this to be part of the payload that we share or upload. For one thing, we could be sharing out-of-date versions of the packages, but even more importantly, we should always be obtaining these from the central npm repository rather from any third party.

Fortunately, the **package.json** file contains all of the information that is needed to re-create our full application. We will demonstrate this by deleting the **node_modules** folder from our application and then running the command

U:\A2> **npm install**

to reinstate them.

Prove that **npm install** rebuilds your application by re-downloading your packages from the npm repository, giving output such as that shown in Figure A2.3 below.



```
wlc-staff-127:A2 adrianmoore$ npm install
a2@1.0.0 /Users/adrianmoore/Desktop/A2
└─┬ express@4.14.0
  ├─┬ accepts@1.3.3
  │ ├─┬ mime-types@2.1.14
  │ │ └── mime-db@1.26.0
  │ └── negotiator@0.6.1
  ├── array-flatten@1.1.1
  ├── content-disposition@0.5.1
  ├── content-type@1.0.2
  ├── cookie@0.3.1
  ├── cookie-signature@1.0.6
  ├─┬ debug@2.2.0
  │ └── ms@0.7.1
  ├── depd@1.1.0
  ├── encodeurl@1.0.1
  ├── escape-html@1.0.3
  ├── etag@1.7.0
  ├─┬ finalhandler@0.5.0
  │ ├── statuses@1.3.1
  │ └── unpipe@1.0.0
  ├── fresh@0.3.0
  ├── merge-descriptors@1.0.1
  ├── methods@1.1.2
  ├─┬ on-finished@2.3.0
  │ └── ee-first@1.1.1
  ├── parseurl@1.3.1
  ├── path-to-regexp@0.1.7
  ├─┬ proxy-addr@1.1.3
  │ ├── forwarded@0.1.0
  │ └── ipaddr.js@1.2.0
  ├── qs@6.2.0
  ├── range-parser@1.2.0
  ├─┬ send@0.14.1
  │ ├── destroy@1.0.4
  │ ├─┬ http-errors@1.5.1
  │ │ ├── inherits@2.0.3
  │ │ └── setprototypeof@1.0.2
  │ └── mime@1.3.4
  ├── serve-static@1.11.1
  ├─┬ type-is@1.6.14
  │ └── media-typer@0.3.0
  ├── utils-merge@1.0.0
  └── vary@1.1.0

npm WARN a2@1.0.0 No repository field.
wlc-staff-127:A2 adrianmoore$
```

*Figure A2.3 Rebuilding the application with **npm install***

A1.1.3 Understanding version numbers

npm packages are identified using the **Semantic Version Specification** (SemVer), which expresses software versions as three values identified as the **Major** version, the **Minor** version and the **Patch** version. Hence our version of Express (**4.16.2**) is Major version **4**, Minor version **16** and Patch version **2**.

In the SemVer scheme, given a version number in the format Major.Minor.Patch, developers should increment the

- MAJOR version when a change results in previous versions of the API being incompatible
- MINOR version when new functionality is added in a backwards compatible manner (i.e. existing software will still work)

- PATCH version when changes are backwards-compatible bug fixes

In other words, a user of a package should always be able to safely upgrade to the latest Patch or Minor version without fear of incompatibility, but upgrading a Major version (i.e. to Express 5.x.x) may cause existing functionality to break and should only be done after thorough testing.

The use of the **^** symbol in the **package.json** entry

```
"express" : "^4.14.0"
```

instructs npm that although it may automatically download the latest Patch and Minor versions as they become available, it should not automatically upgrade to any new Major version that may be released.

Note: See http://semver.org for additional information on Semantic Version Specification

A2.2 Scripts in package.json

As well as providing a representation of the various modules required by our application, **package.json** provide an opportunity for us to specify a range of ways in which we can interact with it.

The "scripts" section in **package.json** contains a range of command specifications that we can use to launch our application. When we created **package.json** by **npm init**, we had the opportunity to specify a test script to be used. Later in the course we will see how this could be used, but as we declined to provide a value, npm generated a default "no test script" entry which can be seen from the code segment below.

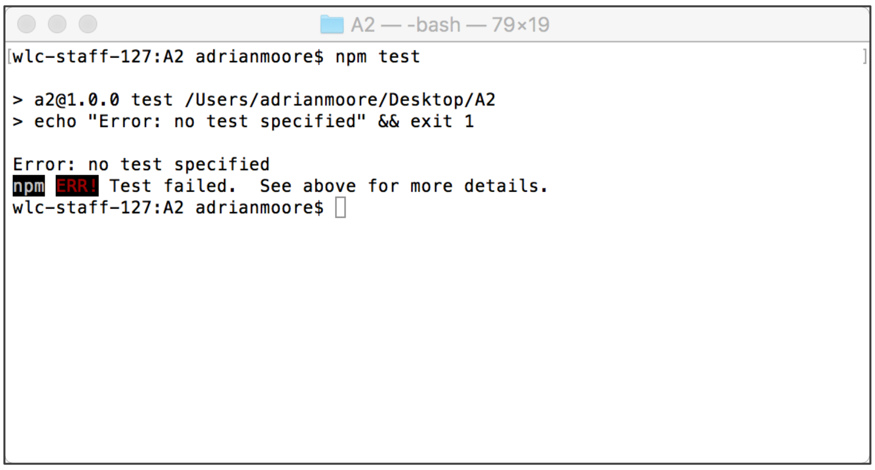
File*: A2/package.json*

```
{
 ...
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
...
}
```

In order to run this script, we simply identify it as a parameter to the **npm** instruction at the command line by issuing

U:\A2> **npm test**

This causes the script to be invoked and provides the output as seen in Figure A2.4 below.



```
wlc-staff-127:A2 adrianmoore$ npm test

> a2@1.0.0 test /Users/adrianmoore/Desktop/A2
> echo "Error: no test specified" && exit 1

Error: no test specified
npm ERR! Test failed.  See above for more details.
wlc-staff-127:A2 adrianmoore$ 
```

Figure A2.4 Running a script

A common use of script is to provide an alternative way of launching our application. Normally, we would launch our program by **node app.js**, so we will provide this command as the body of a new script called **start**.

Add the new script entry as seen in the following code fragment.

File*: A2/package.json*

```
{
 ...
  "scripts": {
    "start" : "node app.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
...
}
```

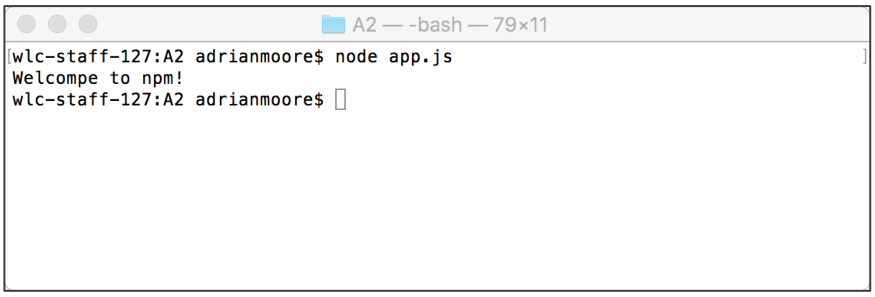
At present, our application doesn't contain an **app.js** file, so let's create one that contains only a very simple **console.log()**.

File: ***A2/app.js***

```
console.log("Welcome to npm!");
```

Now, we can compare the alternative ways of launching our application. First, by requesting node to launch the **app.js** file

U:\A2> **node app.js**



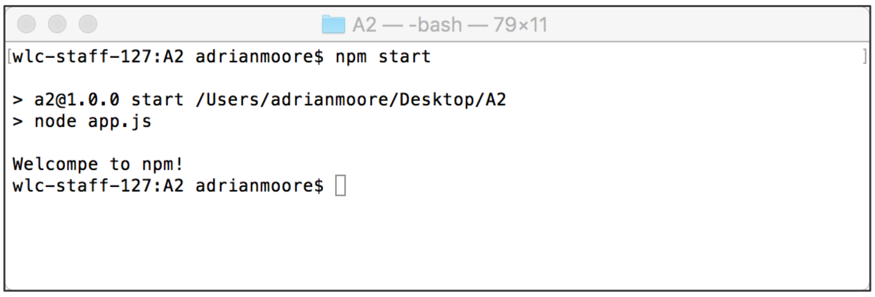
A2 — -bash — 79×11

```
wlc-staff-127:A2 adrianmoore$ node app.js
Welcompe to npm!
wlc-staff-127:A2 adrianmoore$ 
```

*Figure A2.5 Starting an application with a **node** command*

and then by launching our new **start** script

U:\A2> **npm start**



A2 — -bash — 79×11

```
wlc-staff-127:A2 adrianmoore$ npm start

> a2@1.0.0 start /Users/adrianmoore/Desktop/A2
> node app.js

Welcompe to npm!
wlc-staff-127:A2 adrianmoore$ 
```

*Figure A2.6 Starting an application with an **npm** script*

We can see that although the ultimate effect is the same (i.e. the application loads and runs), the **npm start** option provides additional information. This is normally the preferred option for Node developers as it enables us to chain additional commands into the script to retrieve and record other information. For example, we might have an application that behaves differently depending on the availability of an unreliable Internet connection and the script could check the connection status and launch the appropriate Javascript file in response.