Node Package Manager

IMY 220 ● Lecture 16

Contents

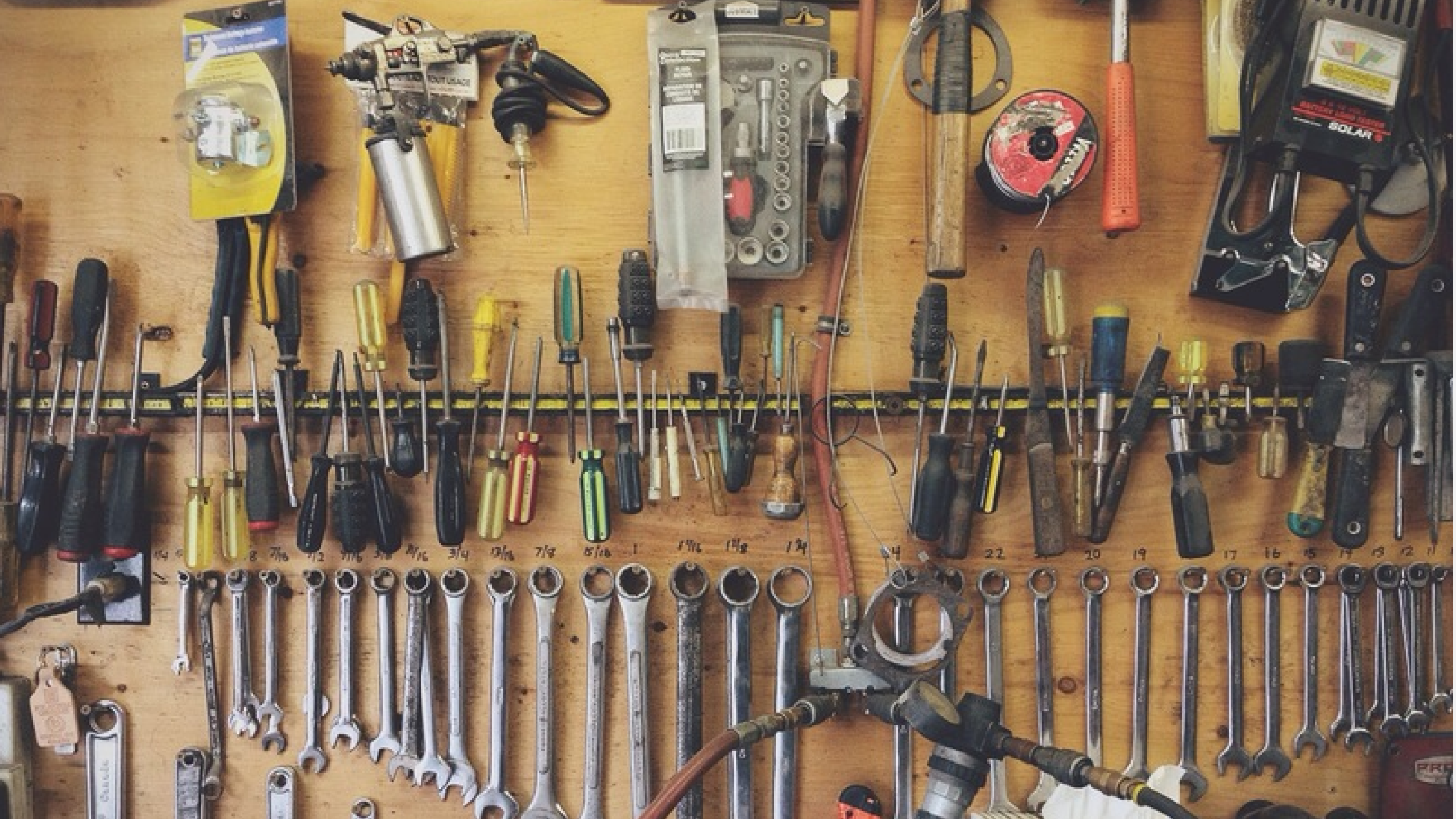
- Node and packages
- npm
- package.json
- Express and Socket.IO

Node.js and packages

When working on large projects, it is always a good idea to organise code into a logical structure (for example, Object Oriented programming)

Node.js (with the help of npm) lends itself towards a modular structure

Relies on many different small bits of reusable code called *packages/modules* (We'll use the terms packages and modules interchangeably in these slides)


SOLAR

Node.js and packages

A package has:

- Some code (that does something)
- A package.json file (that contains metadata about the package)

“The general idea is to create a small building block which solves one problem and solves it well”

https://docs.npmjs.com/getting-started/what-is-npm

This helps to separate concerns and keep your main file clean

Node.js and packages

For example:

In this code, require('http') fetches a built-in package called http which creates a webserver

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Node.js and packages

On top of using built-in packages, Node.js allows you to:

- Easily download and use user-created packages
- Create your own packages

This is done with Node Package Manager (npm)

Node.js and packages

This package-based structure works with Node's *exports* functionality.

For example:

- somePackage.js:

```
module.exports.foo = () => {
      // Do something
}
```

- index.js:

```
let myPackage = require('./somePackage');

myPackage.foo(); // use the package's functionality
```

Node.js and packages

Using this structure, people have created hundreds of thousands of reusable packages that you can download and use through npm

npm currently hosts more than 1 000 000 packages and sees billions of downloads every week

Node.js and packages

Functionally, there is no difference between a node.js project (such as a webserver) and a node.js module (such as a built-in one like ‘http’ or a user-created one like ‘express’)

Any nodejs file(s) that does something and exports that functionality can be used as a module

npm

So what is npm exactly?

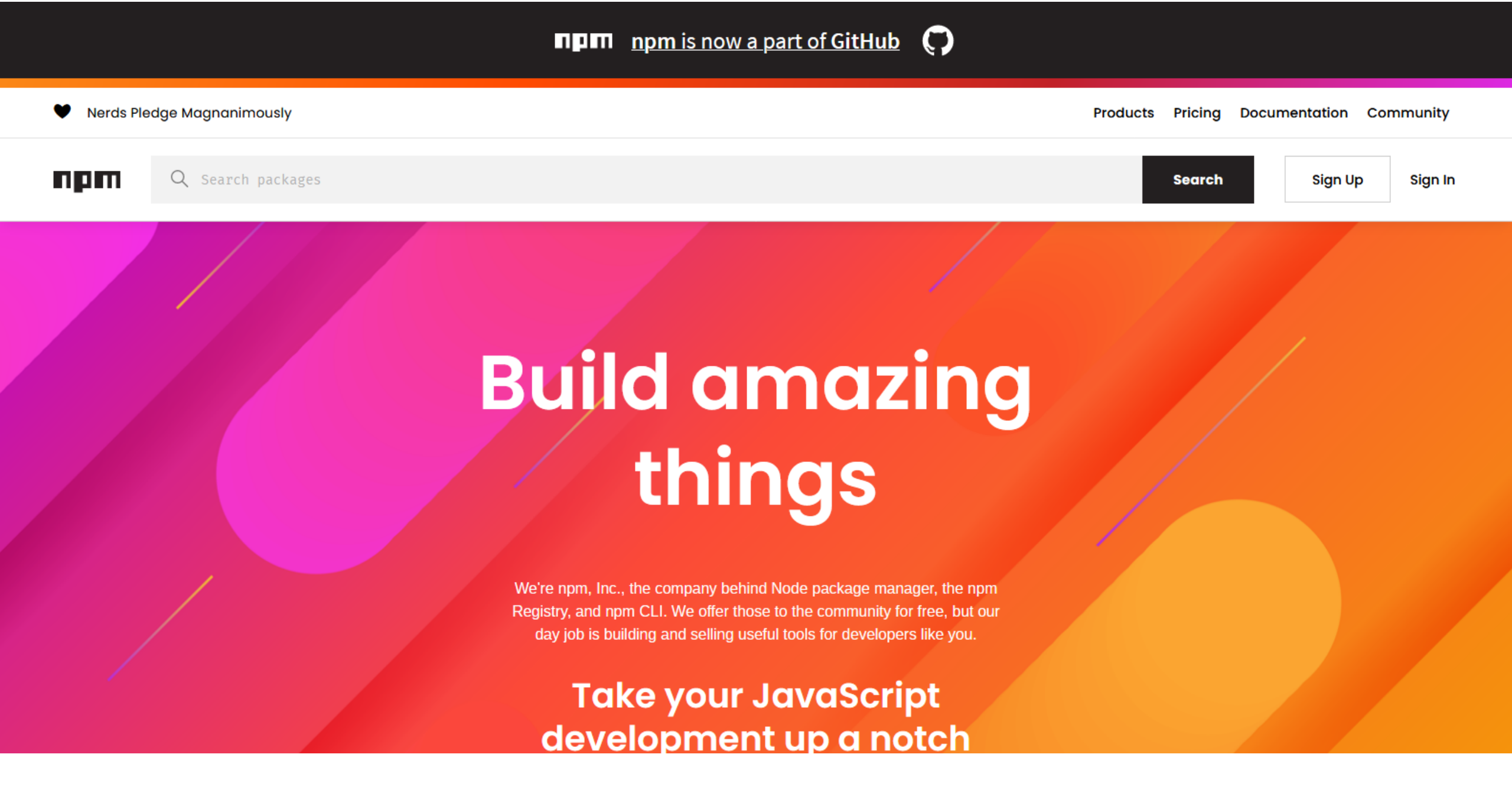
When referring to npm, we can be referring to one of three things:

- The website
- The registry
- The npm-client

npm

The npm-website (https://www.npmjs.com/):

- Lists packages and how to use them
- Each package has a page on npmjs.com that shows you how to install and use that package
- Also includes some general help on how to use npm


npm is now a part of GitHub
Nerds Pledge Magnanimously
Products
Pricing
Documentation
Community
npm
Search packages
Search
Sign Up
Sign In
Build amazing things
We're npm, Inc., the company behind Node package manager, the npm Registry, and npm CLI. We offer those to the community for free, but our day job is building and selling useful tools for developers like you.
Take your JavaScript development up a notch

npm

The npm-registry:

Humongous database with information about packages

Allows consistent access to the different packages (don’t have to go on some dodgy website to download code)

This is used by the **npm-client** when installing packages to find the correct files and so on

npm

The npm-client:

This is the program that runs in the command line that you use to install packages

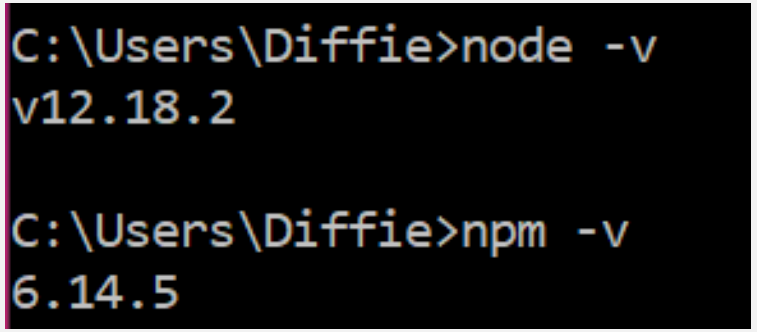
Installed along with node.js (if you downloaded from their website https://nodejs.org/en/)

When you run the command to install packages (which is *npm install <package-name>*) in the command line, the npm-client looks for a package with that package-name in the npm-registry and then downloads the files for that package into your **node_modules** directory (more about this later…)

npm

To check if you have node or npm installed, run *node –v* or *npm –v* in the command line

This returns the version number (if it is installed and the PATH variable is set up correctly)



Installing packages

You can install packages with npm in two ways:

- **Locally**: When you want to use them in your own project/module

- **Globally**: When you want to use them for things like command line tools

We will be focusing on installing packages **locally**

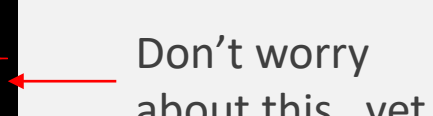
Installing packages locally

To install a package locally, run the *npm install* command (for example, *npm install lodash* to install the package named “lodash”)

- Lodash is just a utility library full of useful JavaScript functions

```
C:\node>npm install lodash
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN node@1.0.0 No description
npm WARN node@1.0.0 No repository field.

+ lodash@4.17.15
added 1 package from 2 contributors and audited 1 package in 1.225s
found 0 vulnerabilities
```



Installing packages locally

Now we can use the package in our own project/module

index.js

```
let _ = require('lodash');

console.log(_.isNumber(5));
```

Set the underscore character to refer to the lodash library (similar to how jQuery functions are assigned to the "$" character).
However, this can be any valid variable name (it doesn't have to be underscore)

Prints out the result of this lodash function

Command line

```
C:\node>node index.js
true
```

Installing packages locally

NB: *npm install <package_name>* requires that you have either a valid *package.json* file or a *node_modules* directory for it to install the package in the right place

If you don't have either of these in your current working directory, npm will keep looking in the parent folder until it finds one of these and then treat that as the "effective working directory" for installing npm modules

Discussion here: https://github.com/npm/npm/issues/3401

Installing packages globally

Installing packages globally works the same, except that you run the command with a *–g* flag after it, which lets you use the package anywhere on your computer

For example: npm install –g nodemon

This would let you use the *nodemon* package in the command line. Packages like CLIs (Command Line Interfaces) are only useful when you install them globally

Installing packages globally

Note that installing packages globally also allows you to use them in any project on your computer

However, this is bad practice

When using a package for a specific project, it should always be installed locally for that project

Uninstalling packages

To uninstall a package (delete the files from your module/project) simply run *npm uninstall <package_name>* in the command line.

```
C:\node\2018\L19>npm uninstall lodash
npm WARN diffie_test@1.0.0 No repository field.

removed 1 package in 1.899s
found 0 vulnerabilities
```

This will remove the files and remove it as a dependency in *package.json* (we'll deal with this next)

Installing packages locally

We can also install many packages simultaneously, for example

npm install lodash express react

And uninstall multiple packages in the same way

package.json

A *package.json* file contains vital/useful information about a module

This is used/modified when installing packages and when running code (for example, to check which packages you've installed, etc.)

package.json

Let's look at what the *package.json* file does and how it works

A typical *package.json* file might look something like this:

```
{
  "name": "diffie_test",
  "version": "1.0.0",
  "description": "A good module. Perhaps even the best.",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Diffie Bosman",
  "license": "ISC",
  "dependencies": {
    "lodash": "^4.17.15"
  },
  "devDependencies": {}
}
```

package.json – name & version

The most important fields in your *package.json* file are *name* and *version*. They're **required** for your package to be installed and used properly.

Both of them have rules and guidelines that can be found here:

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

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

```
{
      "name": "diffie_test",
      "version": "1.0.0",
```

package.json – name & version

The “name” field is the same that is used to require a package

For example:

Express’s package.json:

```
"name": "express",
"repository": {
  "type": "git",
  "url": "git+https://github.com/expressjs/express.git"
},
```

Access package with “name” field’s value

index.js

```
let express = require('express');
```

package.json – description

Next you should have a description field. This contains a brief summary of what your module does.

This is used by the *npm search* on the website to help users find packages

```
"description": "A good module. Perhaps the best.",
```

package.json - main

The “main” field refers to the entry point of your module.

For example, if a module has a file called *index.js* that contains all the module.exports that export the functionality of the module, *main*’s value would be *index.js*

package.json - main

The “scripts” field lets you define scripts to run from the command line

For example, if we wanted to output a message to the console every time our server starts up we could add the following

```
"scripts": {
    "start": "echo 'Starting server' && node index.js"
}
```

Then we can call this script by executing npm start in the console

package.json - main

We can also define scripts and call them as part of other scripts

```
"scripts": {
    "build": "echo 'Starting server'",
    "start": "npm run build && node index.js"
}
```

This is useful for when we have to run tools like Babel or Webpack when starting our server

package.json - dependencies

The “dependencies” field refers to all the modules that are required by a module.

For example, if there is a module that uses other modules to solve some problems…

…instead of having to install that module separately every time someone wants to install this module…

…npm looks at the modules in the *dependencies* field and automatically installs them as part of that module.

package.json - dependencies

Because each package lists its own dependencies and each one of those dependencies can also list its own dependencies…

…installing a large-ish package (such as Express) can lead to installing dependencies that have dependencies that have dependencies and so on…

npm uses the “dependencies” field to automatically take care of this for you

package.json

We're not going to deal with every single possible field in the package.json file (there are quite a few).

You can read up on more fields and where they fit in here:
https://docs.npmjs.com/files/package.json

There's also a nice interactive guide here:
http://browsenpm.org/package.json

package.json

Side note: when you install a package, chances are you'll see something like this:

```
C:\node>npm install lodash
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN node@1.0.0 No description
npm WARN node@1.0.0 No repository field.
```

This is because, to **publish** a module onto the npm-registry it is recommended that you have a description and repository field as part of your package

(If you're not intending to publish a module, you can ignore this)

package.json

Even if you're not planning on publishing your module (or even writing a module), it is still recommended that you create a *package.json* file to keep track of your dependencies and to allow npm's CLI to function properly

You can create a *package.json* file manually by creating the file and writing in the fields or by running *npm init* in the command line and entering the values for the fields given.

```
C:\node>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>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (node) diffie_test
version: (1.0.0)
description: A good module. Perhaps even the best.
git repository:
keywords:
author: Diffie Bosman
license: (ISC)
About to write to C:\node\package.json:

{
  "name": "diffie_test",
  "version": "1.0.0",
  "description": "A good module. Perhaps even the best.",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Diffie Bosman",
  "license": "ISC",
  "dependencies": {
    "lodash": "^4.17.15"
  },
  "devDependencies": {}
}
```

package.json and npm CLI

For older versions of Node, running the command *npm install <package_name>* downloaded the files for a package into the *node_modules* folder

BUT it did not add it to the dependencies list in package.json

To add it to the list, you would have to use the command *npm install <package_name> --save*

The *--save* flag meant that the install command also edited the dependencies in the *package.json* file

package.json and npm CLI

It also worked the same for *uninstalling* packages

So, if you wanted to uninstall a package and remove it from the dependencies list in *package.json*, you would've use the command *npm uninstall <package_name> --save*

(If you want to uninstall a package but not remove it as a dependency, you can still do it with the npm uninstall <package_name> --no-save command)

You can see which packages you have installed by using the *npm ls* command

package.json and npm CLI

You can also install all the packages in your *package.json*'s dependencies list with the *npm install* command

For example, if your package.json has this:

```
"dependencies": {
        "express": "^4.13.3",
        "http-server": "^0.8.0"
},
```

Then running *npm install* will install *express* and *http-server* (will download the files into *node_modules*)

Node.js packages

Next, let’s look at two packages that make it easy to create simple web applications that can communicate between client and server

These packages are:

Express

“*Fast, unopinionated, minimalist web framework for Node.js*”

Backend framework

Allows for easy **routing**

For example: if the user goes to “/”, do this. If the user goes to “/users”, do this…

Also has a scaffolding tool called *express-generator* which quickly generates an application skeleton (which we won’t be looking at)

Express

How to use…

Install with *npm install express*

Remember that you need either a *package.json* file or a node_modules folder in a directory to install packages into that directory

Express

Create a new file, such as ‘index.js’ and paste the following code into it:

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

//CREATE APP
const app = express();

//RESPOND WITH 'HELLO WORLD' FOR ALL REQUESTS FOR THE ROOT URL '/'
app.get('/', (req, res) => {
	res.send('Hello World!');
});

//PORT TO LISTEN TO
app.listen(1337, () => {
	console.log("Listening on localhost:1337");
});
```

Express

Then navigate to *localhost:1337*

You should see the text “Hello World!”

That is because we set up a **route** in Express that responds to all requests for the index page (which is *localhost:1337* in this case) with the text “Hello World!”

Express

You can also use Express to serve static content (such as HTML pages)

```
//CREATE APP
const app = express();

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

This will tell Express to look for an index file (such as *index.html*) in the specified directory (__dirname refers to the current working directory)

So, if you create a file called *index.html*, save it in the same directory as the above code and run the server, you should see the contents of *index.html*

Express

Read more about Express here: http://expressjs.com/

Socket.io

“node.js realtime framework server”

Kind of like jQuery AJAX, but easier

Allows you to easily send messages back and forth between the client(s) and server

Works well with Express

Socket.io

Install with *npm install socket.io*

Again, the --save is optional, but recommended

Include as follows (from http://socket.io/get-started/chat/):

```
const app = require('express')();
const http = require('http').Server(app);
const io = require('socket.io')(http);

http.listen(3000, () => {
      console.log('Listening on *:3000');
});
```

Notice how the constructor is instantiated with an HTTP Server object

The same http-object that is used to instantiate socket.io must be used to start the webserver

Socket.io

What if we wanted to serve static content?

Instead of:

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

Use:

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

Now you can use express.static to serve static content

Socket.io

What if we wanted to use app.listen instead of http.listen as in the example?

Instead of:

```
const http = require('http').Server(app);
const io = require('socket.io')(http);

http.listen(3000, () => {
        console.log('Listening on *:3000');
});
```

Use:

```
const server = app.listen(3000, () => {
        console.log('Listening at localhost:3000');
});

const io = require('socket.io').listen(server);
```

Socket.io

How to send a message from the client to the server.

Client:

```
<script src="/socket.io/socket.io.js"></script>
<script type="text/javascript">
        // This is the socket-object that sends/receives messages in the client
        const socket = io();
        socket.emit('message', 'Hello server');
</script>
```

Server:

```
io.on('connection', socket => {
        socket.on('message', msg => {
                console.log('message: ' + msg);
        });
});
```

This will print out “message: Hello server” in the console when you run the server and go to the webpage

Socket.io

How this works…

http://socket.io/docs/server-api/#socket#emit(name:string[,-…]):socket

```
<script src="/socket.io/socket.io.js"></script>
<script type="text/javascript">
        // This is the socket-object that sends/receives messages in the client
        const socket = io();
        socket.emit('message', 'Hello server');
</script>
```

Name identifier: Sockets with the same name will catch events emitted with this name

Data (Optional): In this case, we're sending through a string, but any data structure is supported, including arrays and JSON objects

Socket.io

The same goes for sending data back to the client from the server

Server:

```
socket.emit('clientMessage', 'Hello client');
```

Client:

```
<script src="/socket.io/socket.io.js"></script>
<script type="text/javascript">
        const socket = io();
        socket.on('clientMessage', message => {
                alert(message);
        });
</script>
```

This will alert “Hello client” when the user goes to the webpage

Socket.io

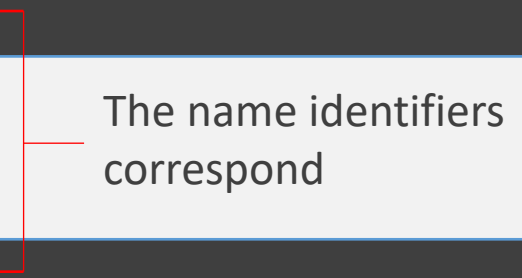
Thus, the general outline for sending message between the client and server looks like this:

Sending messages:

```
socket.emit('name identifier', optionalData);
```

Receiving messages:

```
socket.on('name identifier', optionalData => {
      // Do something with the data
});
```



Socket.io

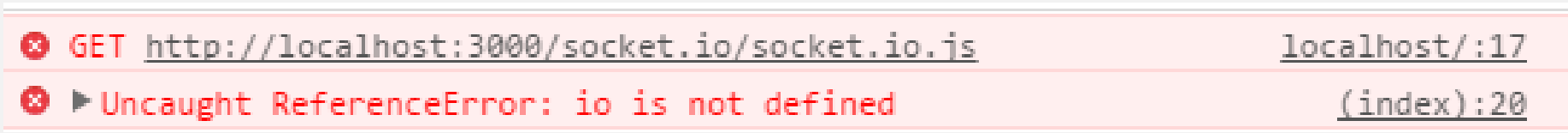
The name identifiers need to be strings that are identical for two or more socket objects to communicate

To emit different messages, simply use different name identifiers (you can use as many as you want)

Socket.io

Quick troubleshoot:

If you get this error in your HTML…



It probably means that your “io”-object isn’t properly instantiated in the server (remember that socket.io needs to be instantiated with an HTTP Server object)

Discussion: http://stackoverflow.com/questions/21365044/cant-get-socket-io-js

Socket.io

The socket.io website has a nice guide for creating a simple chat application using socket.io and Express here: http://socket.io/get-started/chat/

I **highly recommend** that you do this, as it will familiarize you with introductory Node.js, npm, Express and socket.io

Sources

- https://docs.npmjs.com/getting-started/what-is-npm
- https://docs.npmjs.com/files/package.json
- https://www.npmjs.com/package/module-best-practices
- http://browsenpm.org/package.json
- http://expressjs.com/
- http://socket.io/