What is the difference between Bower and npm?

Asked 6 years, 1 month ago Active 2 months ago Viewed 309k times

1714

What is the fundamental difference between `bower` and `npm`? Just want something plain and simple. I've seen some of my colleagues use `bower` and `npm` interchangeably in their projects.

javascript npm bower

464

edited Mar 2 '17 at 18:33
Haifeng Zhang
14.4k 10 44 86

asked Sep 5 '13 at 16:53
Games Brainiac
46.8k 26 117 166

7 Related answer stackoverflow.com/a/21199026/1310070 – sachinjain024 Mar 19 '14 at 12:54

4 possible duplicate of Javascript dependency management : npm vs bower vs volo? – anotherdave Jul 24 '14 at 15:47

7 The answer to this question seems outdated. Can someone tell us what to do in 2016 if we use npm 3 which support flat dependency ? What's the difference wince npm3 and bower and what's the best practice right now ? – amdev Oct 5 '16 at 11:48

2 Bottom-line, @amdev: bower is now deprecated. npm (or Yarn, which is only a slight difference) is where it's at. I'm not aware of any viable alternatives. – XML Jan 7 '18 at 15:32

8 Answers

1889

All package managers have many downsides. You just have to pick which you can live with.

History

npm started out managing node.js modules (that's why packages go into `node_modules` by default), but it works for the front-end too when combined with Browserify or webpack.

Bower is created solely for the front-end and is optimized with that in mind.

Size of repo

npm is much, much larger than bower, including general purpose JavaScript (like `country-data` for country information or `sorts` for sorting functions that is usable on the front end or the back end).

Bower has a much smaller amount of packages.

Handling of styles etc

Bower includes styles etc.

npm is focused on JavaScript. Styles are either downloaded separately or required by something like `npm-sass` or `sass-npm` .

Dependency handling

The biggest difference is that npm does nested dependencies (but is flat by default) while Bower requires a flat dependency tree *(puts the burden of dependency resolution on the user)*.

A nested dependency tree means that your dependencies can have their own dependencies which can have their own, and so on. This allows for two modules to require different versions of the same dependency and still work. Note since npm v3, the dependency tree will by flat by default (saving space) and only nest where needed, e.g., if two dependencies need their own version of Underscore.

Some projects use both is that they use Bower for front-end packages and npm for developer tools like Yeoman, Grunt, Gulp, JSHint, CoffeeScript, etc.

Resources

- Nested Dependencies - Insight into why node_modules works the way it does

edited Aug 1 at 14:02
mike65535
204 2 5 15

answered Sep 6 '13 at 8:09
Sindre Sorhus
59.8k 24 137 192

36 Why doesn't a nested dependency tree do that well on the front end? – Lars Nyström Jan 19 '14 at 16:41

24 Could a front-end npm package not be a flat dependency tree as well? I'm facing the "why do we need 2 package managers?" dilemma. – Steven Vachon Jan 24 '14 at 2:51

37 What do you mean by "flat dependency tree"? Flat tree is what - a list? It's not a tree then. – mvmn Oct 17 '14 at 2:42

14 Actually, a path is also a tree. It is just a special case. From WikiPedia: "In mathematics, and more specifically in graph theory, a tree is an undirected graph in which any two vertices are connected by exactly one path." – Jørgen Fogh Dec 25 '14 at 17:20

42 npm 3 supports a flat dependency tree now. – vasa Nov 21 '15 at 6:55

356

This answer is an addition to the answer of Sindre Sorhus. The major difference between npm and Bower is the way they treat recursive dependencies. Note that they can be used together in a single project.

On the npm FAQ: (archive.org link from 6 Sep 2015)

> It is much harder to avoid dependency conflicts without nesting dependencies. This is fundamental to the way that npm works, and has proven to be an extremely successful approach.

On Bower homepage:

> Bower is optimized for the front-end. Bower uses a flat dependency tree, requiring only one version for each package, reducing page load to a minimum.

In short, npm aims for stability. Bower aims for minimal resource load. If you draw out the dependency structure, you will see this:

npm:

```
project root
[node_modules] // default directory for dependencies
 -> dependency A
 -> dependency B
    [node_modules]
    -> dependency A

 -> dependency C
    [node_modules]
    -> dependency B
      [node_modules]
       -> dependency A
    -> dependency D
```

As you can see it installs some dependencies recursively. Dependency A has three installed instances!

Bower:

```
project root
[bower_components] // default directory for dependencies
 -> dependency A
 -> dependency B // needs A
 -> dependency C // needs B and D
 -> dependency D
```

Here you see that all unique dependencies are on the same level.

So, why bother using npm?

Maybe dependency B requires a different version of dependency A than dependency C. npm installs both versions of this dependency so it will work anyway, but Bower will give you a *conflict* because it does not like duplication (because loading the same resource on a webpage is very inefficient and costly, also it can give some serious errors). You will have to manually pick which version you want to install. This can have the effect that one of the dependencies will break, but that is something that you will need to fix anyway.

So, the common usage is Bower for the packages that you want to publish on your webpages (e.g. *runtime*, where you avoid duplication), and use npm for other stuff, like testing, building, optimizing, checking, etc. (e.g. *development time*, where duplication is of less concern).

Update for npm 3:

npm 3 still does things differently compared to Bower. It will install the dependencies globally, but only for the first version it encounters. The other versions are installed in the tree (the parent module, then node_modules).

- [node_modules]
  - dep A v1.0
  - dep B v1.0
    - ~~dep A v1.0~~ (uses root version)
  - dep C v1.0
    - dep A v2.0 (this version is different from the root version, so it will be an nested installation)

For more information, I suggest reading the docs of npm 3

edited Jul 5 '18 at 13:33
ruffin
10.3k 5 56 103

answered Nov 24 '14 at 13:10
Justus Romijn
12.2k 2 41 55

4 It's almost a cliché now that "software development is all about trade-offs." This is a good example. One must choose *either* greater stability with `npm` *or* minimal resource load with `bower`. – jfmercer Jun 10 '15 at 0:38

6 @Shrek I'm implicitly stating that you actually can use both. They have different purposes, as I state in the final paragraph. It's not a trade-off in my eyes. – Justus Romijn Jun 10 '15 at 8:25

Ahh, I see I misunderstood you. Or I didn't read carefully enough. Thanks for the clarification. :-) It's good that both can be used without a trade-off. – jfmercer Jun 10 '15 at 14:38

4 @AlexAngas I've added an update for npm3. It still has some major differences in comparison to Bower. npm will probably always support multiple versions of dependencies, while Bower does not. – Justus Romijn Jan 18 '16 at 10:13

npm 3 getting closer to bower ;) – ni3 Mar 21 '17 at 6:40

264

TL;DR: The biggest difference in everyday use isn't nested dependencies... it's the difference between modules and globals.

I think the previous posters have covered well some of the basic distinctions. (npm's use of nested dependencies is indeed very helpful in managing large, complex applications, though I don't think it's the most important distinction.)

I'm surprised, however, that nobody has explicitly explained one of the most fundamental distinctions between Bower and npm. If you read the answers above, you'll see the word 'modules' used often in the context of npm. But it's mentioned casually, as if it might even just be a syntax difference.

But this distinction of **modules vs. globals** (or modules vs. 'scripts') is possibly the most important difference between Bower and npm. *The npm approach of putting everything in modules requires you to change the way you write Javascript for the browser, almost certainly for the better.*

The Bower Approach: Global Resources, Like `<script>` Tags

At root, Bower is about loading plain-old script files. Whatever those script files contain, Bower will load them. Which basically means that Bower is just like including all your scripts in plain-old `<script>`'s in the `<head>` of your HTML.

So, same basic approach you're used to, but you get some nice automation conveniences:

- You used to need to include JS dependencies in your project repo (while developing), or get them via CDN. Now, you can skip that extra download weight in the repo, and somebody can do a quick `bower install` and instantly have what they need, locally.
- If a Bower dependency then specifies its own dependencies in its `bower.json`, those'll be downloaded for you as well.

But beyond that, *Bower doesn't change how we write javascript*. Nothing about what goes inside the files loaded by Bower needs to change at all. In particular, this means that the resources provided in scripts loaded by Bower will (usually, but not always) still be defined as *global variables*, available from anywhere in the browser execution context.

The npm Approach: Common JS Modules, Explicit Dependency Injection

All code in Node land (and thus all code loaded via npm) is structured as modules (specifically, as an implementation of the CommonJS module format, or now, as an ES6 module). So, if you use NPM to handle browser-side dependencies (via Browserify or something else that does the same job), you'll structure your code the same way Node does.

Smarter people than I have tackled the question of 'Why modules?', but here's a capsule summary:

- Anything inside a module is effectively *namespaced*, meaning it's not a global variable any more, and you can't accidentally reference it without intending to.
- Anything inside a module must be intentionally injected into a particular context (usually another module) in order to make use of it
- This means you can have multiple versions of the same external dependency (lodash, let's say) in various parts of your application, and they won't collide/conflict. (This happens surprisingly often, because your own code wants to use one version of a dependency, but one of your external dependencies specifies another that conflicts. Or you've got two external dependencies that each want a different version.)
- Because all dependencies are manually injected into a particular module, it's very easy to reason about them. You know for a fact: *"The only code I need to consider when working on this is what I have intentionally chosen to inject here"*.
- Because even the content of injected modules is *encapsulated* behind the variable you assign it to, and all code executes inside a limited scope, surprises and collisions become very improbable. It's much, much less likely that something from one of your dependencies will accidentally redefine a global variable without you realizing it, or that you will do so. (It *can* happen, but you usually have to go out of your way to do it, with something like `window.variable` . The one accident that still tends to occur is assigning `this.variable` , not realizing that `this` is actually `window` in the current context.)
- When you want to test an individual module, you're able to very easily know: exactly what else (dependencies) is affecting the code that runs inside the module? And, because you're explicitly injecting everything, you can easily mock those dependencies.

To me, the use of modules for front-end code boils down to: working in a much narrower context that's easier to reason about and test, and having greater certainty about what's going on.

It only takes about 30 seconds to learn how to use the CommonJS/Node module syntax. Inside a given JS file, which is going to be a module, you first declare any outside dependencies you want to use, like this:

```
var React = require('react');
```

Inside the file/module, you do whatever you normally would, and create some object or function that you'll want to expose to outside users, calling it perhaps `myModule`.

At the end of a file, you export whatever you want to share with the world, like this:

```
module.exports = myModule;
```

Then, to use a CommonJS-based workflow in the browser, you'll use tools like Browserify to grab all those individual module files, encapsulate their contents at runtime, and inject them into each other as needed.

AND, since ES6 modules (which you'll likely transpile to ES5 with Babel or similar) are gaining wide acceptance, and work both in the browser or in Node 4.0, we should mention a good overview of those as well.

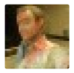
More about patterns for working with modules in this deck.

EDIT (Feb 2017): Facebook's Yarn is a very important potential replacement/supplement for npm these days: fast, deterministic, offline package-management that builds on what npm gives you. It's worth a look for any JS project, particularly since it's so easy to swap it in/out.

EDIT (May 2019) "Bower has finally been deprecated. End of story." (h/t: @DanDascalescu, below, for pithy summary.)

And, while Yarn is still active, a lot of the momentum for it shifted back to npm once it adopted some of Yarn's key features.

edited May 11 at 17:41 answered Jul 26 '15 at 3:12


XML
16.6k 7 58 63

13 Glad this answer was here, the other popular answers don't mention this detail. npm forces you to write modular code. – Juan Mendes May 13 '16 at 13:06

I'm sorry, from a folk that cares very little for all the fuzzing in the javascript parlands but so happens it runs a business that makes use of a small web application. Recently been forced to try npm, from using bower with the toolkit we use to develop the darn web thing. I can tell you that the biggest difference is wait time, npm takes ages. Remember that is compiling xkcd cartoon with the guys playing sword fights yelling 'compiling' to their boss; thats pretty much what npm added to bower. – Pedro Rodrigues Nov 21 '18 at 21:00

2017-Oct update

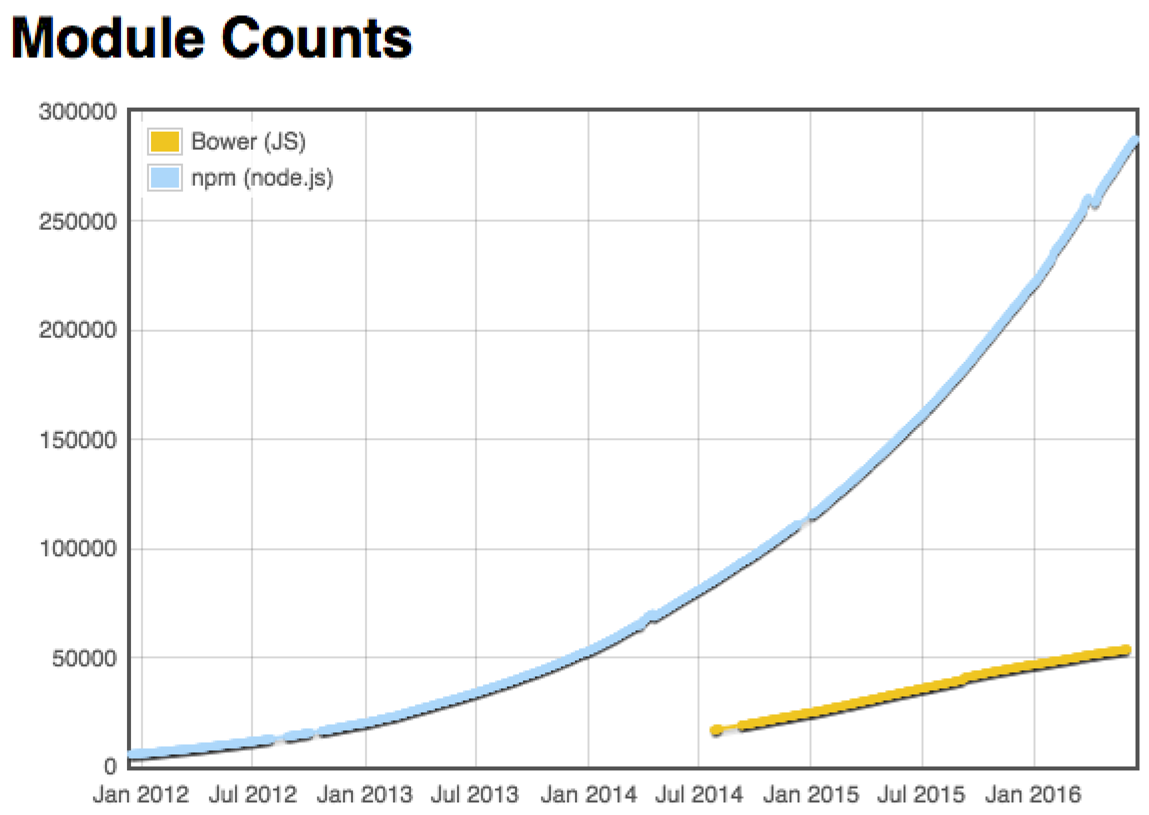
127

Bower has finally been deprecated. End of story.

Older answer

From Mattias Petter Johansson, JavaScript developer at Spotify:

> In almost all cases, it's more appropriate to use Browserify and npm over Bower. It is simply a better packaging solution for front-end apps than Bower is. At Spotify, we use npm to package entire web modules (html, css, js) and it works very well.
>
> Bower brands itself as the package manager for the web. It would be awesome if this was true - a package manager that made my life better as a front-end developer would be awesome. The problem is that Bower offers no specialized tooling for the purpose. It offers NO tooling that I know of that npm doesn't, and especially none that is specifically useful for front-end developers. **There is simply no benefit for a front-end developer to use Bower over npm.**
>
> We should stop using bower and consolidate around npm. Thankfully, that is what is happening:



> With browserify or webpack, it becomes super-easy to concatenate all your modules into big minified files, which is awesome for performance, especially for mobile devices. Not so with Bower, which will require significantly more labor to get the same effect.
>
> npm also offers you the ability to use multiple versions of modules simultaneously. If you have not done much application development, this might initially strike you as a bad thing, but once you've gone through a few bouts of Dependency hell you will realize that having the ability to have multiple versions of one module is a pretty darn great feature. Note that npm includes a very handy dedupe tool that automatically makes sure that you only use two versions of a module if you actually *have* to - if two modules both *can* use the same version of one module, they will. But if they *can't*, you have a very handy out.

(Note that Webpack and rollup are widely regarded to be better than Browserify as of Aug 2016.)

edited Nov 16 '17 at 21:40

answered Jul 4 '15 at 10:48

Dan Dascalescu
78.5k 23 221 300

1 webpack and npm is even better I think.. – refactor Aug 30 '16 at 13:58

6 <sarcasm> Please have in mind that even 'hello world' npm project needs 300+ modules to run...</sarcasm> :O – Mariusz Jamro Dec 11 '16 at 18:36

I don't agree that "big minified files" are "awesome for performance, especially for mobile devices". Quite the opposite: Restricted bandwidth requires small files, loaded on demand. – Michael Franzl Apr 9 '17 at 14:03

Not very good advice. Most npm packages are nodejs backend only. If you are not doing javascript on the backend, or you dont have a module system in place, the number of packages is irrelevant because Bower will fit your needs much better – Gerardo Grignoli Apr 11 '17 at 17:59

4 @GerardoGrignoli: bower is on its way out. – Dan Dascalescu Apr 12 '17 at 8:30

45

Bower maintains a single version of modules, it only tries to help you select the correct/best one for you.

> Javascript dependency management : npm vs bower vs volo?

NPM is better for node modules because there is a module system and you're working locally. Bower is good for the browser because currently there is only the global scope, and you want to be very selective about the version you work with.

edited May 23 '17 at 11:55
Community ♦
1 1

answered Jul 19 '14 at 20:59
Sagivf
575 5 9

4 I feel that Sindre mentions that when he talks about nested dependency. – Games Brainiac Jul 19 '14 at 21:21

5 @GamesBrainiac your correct, just thought i'd put it in my own words. – Sagivf Jul 20 '14 at 11:01

1 @Sagivf These are ***NOT*** your own words, unless you are also wheresrhys who provided the original answer here – dayuloli Mar 3 '15 at 6:36

4 @Sagivf There's nothing wrong with copying ***relevant parts* of other's answers if they didn't provide an answer here themselves. It just bugged me a little you said "just thought i'd put it in my own words." Credit should go where credit is due. – dayuloli Mar 4 '15 at 10:39

2 I don't know why you guys picked on this answer so much. There is indeed new information/perspective in this answer to me. – Calvin Nov 30 '15 at 22:04

My team moved away from Bower and migrated to npm because:

33

- Programmatic usage was painful
- Bower's interface kept changing
- Some features, like the url shorthand, are entirely broken
- Using both Bower and npm in the same project is painful
- Keeping bower.json version field in sync with git tags is painful
- Source control != package management
- CommonJS support is not straightforward

For more details, see "Why my team uses npm instead of bower".

edited Nov 21 '15 at 4:04
user2864740
46.1k 8 87 158

answered Feb 16 '15 at 21:04
Nick Heiner
49.8k 164 430 669

17

Found this useful explanation from http://ng-learn.org/2013/11/Bower-vs-npm/

> On one hand npm was created to install modules used in a node.js environment, or development tools built using node.js such Karma, lint, minifiers and so on. npm can install modules locally in a project (by default in node_modules) or globally to be used by multiple projects. In large projects the way to specify dependencies is by creating a file called package.json which contains a list of dependencies. That list is recognized by npm when you run npm install, which then downloads and installs them for you.
>
> On the other hand bower was created to manage your frontend dependencies. Libraries like jQuery, AngularJS, underscore, etc. Similar to npm it has a file in which you can specify a list of dependencies called bower.json. In this case your frontend dependencies are installed by running bower install which by default installs them in a folder called bower_components.
>
> As you can see, although they perform a similar task they are targeted to a very different set of libraries.

edited Oct 10 '14 at 3:26

answered Oct 3 '14 at 0:08
Henry Neo
2,227 1 20 27

1 With the advent of `npm dedupe` , this is a bit outdated. See Mattias's answer. – Dan Dascalescu Jul 4 '15 at 10:49

7

For many people working with node.js, a major benefit of bower is for managing dependencies that are not javascript at all. If they are working with languages that compile to javascript, npm can be used to manage some of their dependencies. however, not all their dependencies are going to be node.js modules. Some of those that compile to javascript may have weird source language specific mangling that makes passing them around compiled to javascript an inelegant option when users are expecting source code.

Not everything in an npm package needs to be user-facing javascript, but for npm library packages, at least some of it should be.

edited Oct 11 '14 at 21:27

answered Oct 11 '14 at 20:42
jessopher
102 1 6

This npmjs blog post states "Your package can contain anything, whether it's ES6, client-side JS, or even HTML and CSS. These are things that naturally turn up alongside JavaScript, so put them in there.". – Dan Dascalescu Jul 4 '15 at 10:28

1 There is a difference between *can contain*, and *should include*. Of course they can contain anything, but in general, they *should include* some sort of interface to commonJS. It is the 'node package manager' after all. The part about *These are things that naturally turn up **alongside Javascript*** is important. There are lots of things that are tangentially related to javascript that do not *naturally turn up along side* it. – jessopher Dec 9 '15 at 2:43

protected by Games Brainiac Feb 14 '15 at 12:31

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?