

Sane Dependencies

Develop, Publish, Deploy

Tuesday, January 17, 12

hi hi hi

I'm isaac, one of the node.js people.

I'm here to talk to you about dependencies.

node.js

Tuesday, January 17, 12

I work on Node.js at Joyent.

How many people here write node programs?

How many of you use npm to install node programs?

Probably don't need to go into too much history here. (just really quickly...)

story time

Tuesday, January 17, 12

I'd like to just explain how some of these ideas came about.

I'm going to gloss over a lot of details, because I'd like to focus on what's practical, but I do want to convey some of the depth and history of this.

When I started with Node, it was an exciting time. It still is exciting. A lot of people were writing really interesting programs, and it was always really tricky to get them installed just right. A lot of people mentioned that we needed something like CPAN or Rubygems or Pear for all these node programs.

I'd used a program called yinst quite a bit at Yahoo.

yinst is awesome

Tuesday, January 17, 12

and let me tell you, it's awesome. As I'd been a php developer, I'd also had some experience with Pear

pear sucks

Tuesday, January 17, 12

which I found to be pretty much intolerable. It just doesn't seem to get people excited, there's very little enthusiasm or interest in it.

Around the same time, there was this CommonJS thing,

CommonJS

Tuesday, January 17, 12

and there was some momentum behind Tusk and some other platforms. Some folks thought we could make a specification, and then agree on it, and then have implementations that could be swapped out easily.

That... didn't work out so well. As it happens, getting people to agree is really costly, and then you're usually stuck with a bunch of decisions that were the most agreeable, instead of being the best decisions, and you almost never know ahead of time what's going to work out the best.

But, one artifact of that did work out really well, and that's having a single json file in your program that defines what it is.

package.json

Tuesday, January 17, 12

I didn't come up with this pattern, credit for this goes to Narwhal's Tusk program.

The main bits are name and version:

```
$ cat package.json
```

```
{ "name": "foo"  
  , "version": "1.2.3"  
  , "description": "foo in js!"  
  , "main": "foo.js"  
  , "dependencies":  
    { "bar": "2.5.1" }  
}
```

Tuesday, January 17, 12

So, you had this one artifact that could be used to state declaratively what your program IS, and what it NEEDS to function, and even a little hint of HOW to use it.

kinda awesome.

Tuesday, January 17, 12

This is a pretty big deal, actually. Of course, it's not that different from a gemfile, but the fact that it was json rather than javascript meant that it could not execute code, and thus it forced us into this very declarative worldview.

Also, that means you can shove it in a nosql document store, or save it to disk, or load it in a program object, and it's very nice and easy to work with. JSON isn't the prettiest format in the world, but it's the prettiest format that has a clear and obvious 1:1 relationship to a JavaScript object, which is a huge win.

So, what's a dependency?

Tuesday, January 17, 12

This was not always so obvious a question. Dig through the CommonJS mailing list, there are ashes of many many heated discussions about this.

dependency: *noun*
one that is relied on

Tuesday, January 17, 12

Well, of course, a dependency is something you depend on.

```
"dependencies":  
{  
  "bar": "2.5.1"  
  , "os": ["darwin", "linux"]  
  , "http":  
    {  
      "curl": "7.21"  
      , "wget": "1.13.*"  
      , "request": "2.x"}  
  , "remote":  
    {  
      "type": "git",  
      , "href": "ssh://x:foo.git"}  
}
```

Tuesday, January 17, 12

That's not very specific

i'm sure you can imagine, it took a bit of work to figure out what we're really talking about.

Eventually, we came to a somewhat sane understanding that a "dependency" of a package is another package, with a version designator of some sort.

dependency *noun*
an installable package
that another installable
package uses.

Tuesday, January 17, 12

That is, when we talk about "dependencies", we're not specifying the operating system, or what might be installed somewhere else on the system. We're specifically talking about the same **kind** of thing that the package.json is describing.

So, how do you specify your dependencies? Well, they all have a name and a version. But how do we actually get from that to a working program?

Easy! Just use semver!

Tuesday, January 17, 12

Anyone who ever suggests that you can "just" use semver hasn't actually tried it with more than 1 person. I like to think of the word "just" (as a huge red flag.)

Easy! **Just** use semver!

Tuesday, January 17, 12

I like to think of the word "just" as a huge red flag. It means that complexity is trying to hide from you.

It's not that the concepts expressed on semver.org aren't good ideas. They're great. It's just that that's only half the story.

So, briefly, what is semver anyway?

SEMantic VERsioning

- X.Y.Z
- X.Y.Ztag
- X.Y.Z-3-tag

Tuesday, January 17, 12

The idea of semver is that you bump the Z value, the "patch" version any time you release any kind of bugfix that doesn't change the API. In theory, it's always safe for users to upgrade to a new patch release.

You change the Y value, the "minor" version, if there's an API addition or modification, but the semantics of the program are roughly the same, and existing programs "should" still work. (Should is naother red flag.)

You bump the X value, the "major" version, whenever there is a fundamental change to the architecture or semantics of the program.

Of course, in real life, you dont' know that a bugfix won't break some internal untested edge case that caused someone else's program to work by accident, even though it wasn't documented or anything.

"foo": "1.2.3"

- foo 1.2.3, 1.3.0, but not 2.0.0
- foo 1.2.3, 1.2.4, but not 1.3.0
- foo 1.2.3 only
- (there's a right answer to this)

Tuesday, January 17, 12

I think we've learned from git that, two things:

a) tools are best when they don't try to be too clever,
and b) you have to be really really careful to not make your tools too clever, because it so often seems like such a good idea.

So, with semver, it's tempting to say that if you depend on 1.2.3, that I can give you 1.2.4 instead, and that should be safe.

use semver.
don't abuse semver.

Tuesday, January 17, 12

If you say 1.2.3, a tool should give you exactly 1.2.3, or fail. and then if you do want ranges,
(there are ways to get that)

ranges

- $\geq 1.2.3 < 1.2.5$
- $1.2.3 \parallel 1.2.6 \parallel \geq 2.3.0 < 2.5.0$
- $2.x$
- $\sim 3.4.6$
- ~ 4.7

Tuesday, January 17, 12

...there are ways to do that.

The struggle so far has been to make each of these mean what people think it means. There are a bunch of edge cases (and one notable footgun)

`>=` considered harmful

Tuesday, January 17, 12

...and one notable footgun.

Here's the thing, npm is an idiot. It's a computer program. It's not a person, and it's not smart (even though it does love you).

So, when you say "`>=1.0.2`", and version 2.0 comes out that changes everything, it's going to do what you tell it, and go ahead and try to work with version 2.0, and someone's going to get bitten by a bug.

This isn't the end of the world. But, specifying a version range that your program works with, that's sort of a promise. Using `>=` is a much bigger promise than using `~`.

~ is much better

- ~2.3.4 is $\geq 2.3.4 < 2.4.0$ -
- ~2.3 is $\geq 2.3.0 < 3.0.0$ -
- ~2 is $\geq 2.0.0 < 3.0.0$ -
- It's basically the "be semver-reasonable" operator.

Tuesday, January 17, 12

So what's ~ mean?

Well, it's basically a way of saying, "I know how this package author versions their stuff, and I know what kinds of changes are going to break me and what kinds aren't, so go ahead and do the semver thing"

It's a nice compromise.

Now, I'd be remiss if I didn't mention (dependency hell)

Dependency Hell

- A requires B@1.2.3
- C requires B@2.3.4
- D requires A and C
- Conflict!

Tuesday, January 17, 12

...dependency hell

As software communities grow, if they are doing anything, you invariably find cases where you try to install something, and it has a conflict in its dependency tree.

The costs of these conflicts aren't just in the time spent to sort it out, but also in the ideas and inspiration lost, because someone was afraid to change something or do it a new way

unacceptable.

Tuesday, January 17, 12

This was intolerable to me. npm was explicitly a way to make it so that these kinds of conflicts were never exposed to users. we'd trade that for having more than one version of something installed, side by side, and just handle it. after all, programs aren't *that* big, having two copies isn't terrible.

I'd dealt with dependency hell a lot in the past, and it always seemed like there should be a better way

so, in a fit of hubris, (I did a very bad thing)

npm 0.x

- single root folder
- `/path/to/foo@1.2.3`
`/path/to/bar@2.1.5`
- Symlink the "active" version
- Symlinks and shims everywhere

Tuesday, January 17, 12

I did a very bad thing, and in a fit of hubris, wrote the first few versions of npm.

The idea was that you'd just kind of do `require("foo")``, and get the right version.

npm 0.x

- debugging nightmare
- Could never be windows compliant
- reliance on require.paths that didn't actually work in all cases.
- shims EVERYWHERE.
- `require("foo@1.3.4")` `<==` problematic.

Tuesday, January 17, 12

This used a bunch of shim files and symlinks to sort of make it work. There were things that weren't supposed to be used, but of course, the minute you expose an implementation detail, someone's gonna start depending on it, and now you're kind of screwed.

So, this was not so good.

npm 1.x

- Bundle everything
- Nested node_modules folders
- Conflicts and cycles via shadowing
- No shims*

** except for executable scripts on windows, since it lacks #! support.*

Tuesday, January 17, 12

The way of npm 1.0 is much nicer. Packages are a lot more organized, you can even put them in place with git submodules if you like, explore them easily, since they're in a predictable place, and so on.

So, with the history lesson out of the way, (let's talk about actually using these things)

The part where I tell
you what to do.

Tuesday, January 17, 12

let's talk about actually using these things. There are three very different subjects that I want to touch on.

develop

Tuesday, January 17, 12

Programs start with development, obviously. that's a very important phase of their life.

To a large extent, how you develop a thing is somewhat determined by what you're going to end up doing with it.

publish

Tuesday, January 17, 12

you might be developing a reusable module that you're going to share with others. If you're using node, then that probably involves the ``npm publish`` command.

Usually, if you're publishing something, that means it's a library or utility that you expect people to install on their systems, and use. so, they're going to list it as a dependency.

deploy

Tuesday, January 17, 12

a very different use-case is when you want to deploy something, like a website, or a downloadable application.

In this case, you're **not** interested in having other people install your stuff. What you want is to have a site that can stand up and serve requests. In fact, you almost certainly are not going to be publishing the whole site to any kind of package registry, since it only is intended to live in one place.

thumb rules

- Use "~3.0" or "3.x" instead of ">=3.0.0"
- Libraries: Flexible
- Apps/deployments: Strict

Tuesday, January 17, 12

Here's a few rules of thumb

First, I said it before, I'll say it again, don't use `>=` version ranges. Use `~` or `.x` versions. Make promises you can keep.

Libraries should be as flexible as reasonable in their dependencies.

Apps should be as strict as possible.

libraries: as flexible as reasonable

Tuesday, January 17, 12

If you're building a reusable component, you probably want to be able to benefit from bug fixes and improvements in your dependencies. And, since you don't really need to predict every possible environment where it's going to run, what's really important is the code contract that you set up, and the best way to verify that contract is by having (good tests)

have good tests

Tuesday, January 17, 12

by having good tests.

When those tests fail, you'll want to make sure that youv'e got (a place to post issues and send patches)

list your github repo

and use it.

Tuesday, January 17, 12

that you've got a place to post issues and send patches

If you list your github repo in your package.json file, your users can use the `npm bugs` and `npm docs` commands to open up the appropriate page in their web browser.

Bundling deps for reusable modules: sub-optimal

unless you really need to for some reason

Tuesday, January 17, 12

bundling the dependencies for a reusable node module is not really ideal. there are a few reasons for this.

why not to bundle libraries' deps:

- Having users test against multiple versions of dependencies makes for better coverage
- Increases the number of use cases and situations that your code is exposed to
- More flexible => more users => more patches => more fame => more awesome
- Free bug fixes when deps update!

Tuesday, January 17, 12

with a reusable module, a library or something that depends on some other library, you really want to make it as likely as possible that you'll benefit from an update in another part of the code. You should still not use `>=` ranges, but `~` or `.x` is usually ideal.

This increases the areas that your code ends up being used in, resulting in more and better feedback. It reduces the likelihood that your program will need to have its own copy of any dependencies, instead using whatever is already there in the user's environment if possible, and so on.

Also, (this is easier)

also, this is easier

Tuesday, January 17, 12

this is easier.

It's easier to install your thing, you can do `npm update` to get new versions of your dependencies to test against, and so on.

People tend to dramatically underestimate the costs of friction. In practice, even a small reduction in friction can result in a huge increase in productivity, and can lead to new ideas that you wouldn't have even considered before.

making development easy isn't a minor thing. it's what we need to do for ourselves, and for everyone who uses the tools we create. Making it easy, isn't a footnote -- it's **the point**. It's **why we're doing this**.

apps: be ludicrously strict

Tuesday, January 17, 12

If you're deploying something, if you're uploading it to a server and it's gonna run your website, or if it's an application that's installing on someone's computer or device, and they're not developing a program that interacts with it programmatically, it's just the opposite.

In this case, what you really want is for a given build to be 100% reproducible. If there's a bug, you want to be able to figure out exactly what change caused it.

Thankfully, we have a great tool for locking down dependencies. I'm not talking about npm.

```
npm install foo
git add node_modules
git ci -m "installed foo"
```

Tuesday, January 17, 12

i'm talking about git.

<beat>

Seriously. Do this. If you're deploying a thing, if you're having users download an app that you built, and you want to be able to keep yourself from going crazy at the worst possible time because a build is not working, you've gotta be tracking your dependencies in git.

seriously.

Tuesday, January 17, 12

seriously

there are a lot of benefits to checking your node_modules folder into git.

Benefits

- `git bisect` is super useful.
- replay the same build again later
- npm is not a content tracker
- if you don't, you'll wish you did later

Tuesday, January 17, 12

it means basically that a build or a deployment is a specific snapshot of your application, and that you can deterministically run a build with confidence, because (it's all in the repository)

it's all in the repository

Tuesday, January 17, 12

it's all in the repository.

This is the simplest way to have stable, debuggable, predictable, reproducible builds of a deployable application. Bundle your dependencies, check them into source control, deploy based on a specific commit sha (tag, whatever)

The most common objection to this simple, obvious fact:

clutter up my git history?
ew.

Tuesday, January 17, 12

Somehow, there's a misconception that it's going to add a lot of clutter to your git history, in fact, (no, it doesn't)

No, it doesn't.

- Your dependencies are **a part of your program.**
- There's no way around this fact. You're deploying them, so you need to track them.
- What you have is a single "update foo" commit here and there. It's a change, why not track it in the change tracker?

Tuesday, January 17, 12

no, it doesn't.

The simple fact of the matter is, if you're deploying your dependencies (and you are), then they're part of your program. You have to track them.

Another common response:

*why not just lock your
dependency versions?*

Tuesday, January 17, 12

lock the dependencies to a specific version, rather than a range.

There are two problems with this.

problem one: that's a pita.

Tuesday, January 17, 12

the first problem is that that's kind of a pain in the ass for development.

it means that each module you create is another thing that must be updated in multiple places for every change.

So, you have to change a json file, **check that file into git**, then install the new version of the thing that matches that version. You end up with the same number of commits, but they're more of a pain, and instead of just running a command, you have to edit a file **and** run a command!

All this increases friction along the path of small composable modules, and decreases friction along the path of "giant monolithic pile of code". So, guess which one you tend to end up with.

problem two: dependencies have dependencies

Tuesday, January 17, 12

And, even if you lock **your** dependencies to a specific version, you don't necessarily know if they're going to lock **their** dependencies to a specific version, meaning that you can **still** end up with an unpredictable state when you deploy.

This is not good.

what about .gitignore files?

Tuesday, January 17, 12

sometimes, dependencies have .gitignore files in them, and this traditionally was a bit of a snag for users who checked deps into git

thankfully, (this is fixed now.)

.gitignore no more!

Tuesday, January 17, 12

this is fixed now. npm renames gitignore files to .npmignore if there's no .npmignore file already, and excludes them from packages it installs and publishes.

This is pretty new, in 1.1. But moving forward, that's how it's going to look. You shouldn't end up with .gitignore files in a package any more.

What about compiled dependencies?

Tuesday, January 17, 12

This is actually a pretty valid concern. If you depend on something like expat or libpng, then it's going to need to be compiled on the target architecture. If you're using a mac, but deploying to linux or smartos, then that means you have to re-compile.

Thankfully, there's a command for that:

npm rebuild

Tuesday, January 17, 12

npm rebuild has been around almost as long as `install`. It does what you think.

So, to recap:

Ideal deployifier

- check everything into git.
- send code from git repo to target location
- `npm rebuild`
- `npm start`

Tuesday, January 17, 12

the ideal deploy tool would just take whatever i've got in git, node_modules and all
put it in the proper place

run npm rebuild to make sure it's compiled for the target architecture

then run `npm start` to do whatever it is that starts my app

more winful

- new users check out the app repo, run `npm rebuild`, and they're ready to go
- no http fetch at build time, one fewer moving part that can fail
- make modifications to a dependency
- deploy with a dep that isn't published

Tuesday, January 17, 12

There are other hidden benefits.

New developers on your team can hit the ground running. Check out the app, npm rebuild, and they're in business.

No http fetch at build time, which is one fewer moving part, faster deploys and so on.

You can deploy with a dependency that hasn't been published, maybe something you're working on, or something you've modified for some reason.

\(ToT\)

Tuesday, January 17, 12

unfortunately...

this is 100% impossible
on heroku today

Tuesday, January 17, 12

there's no way to deploy this way on heroku, since it actually deletes your node_modules folder out of the git working directory

[https://github.com/
heroku/heroku-buildpack-
nodejs/issues/6](https://github.com/heroku/heroku-buildpack-nodejs/issues/6)

Update: David Dollar saw my talk and fixed it promptly.

Tuesday, January 17, 12

i've posted an issue. I took a look at the code, but the best way to make the change wasn't obvious, or I would have been happy to send a pull request instead.

(/ToT)/

Tuesday, January 17, 12

also...

no.de supports bundled
modules, but doesn't
rebuild them.

Edit: It does now.

Tuesday, January 17, 12

Joyent's no.de system supports bundled node_modules folders, but doesn't rebuild them.
(same for nodejitsu)

nodejitsu is the same

Edit: Nodejitsu respects bundled deps if they're in the "bundledDependencies" array in package.json.

Tuesday, January 17, 12

same for nodejitsu

there's work to be done

Tuesday, January 17, 12

So, there are some improvements yet to be made in the area of deploying node programs.

Most of the large-scale node deployments are currently using or coming around to some home-grown combination of tools involving git, bundled dependencies, and rebuild scripts. But there's nothing for the general public, and no general purpose library or tool that makes all this easy

talk to me

i@izs.me

<http://twitter.com/izs>

<http://github.com/isaacs>

Tuesday, January 17, 12

And, I think we're just about out of time, but please send me a message any which way, or come grab me this afternoon. I love to chat about this, so it'd be great to hear any ideas or experience you've had.