TINK

A NEXT-GENERATION PACKAGE MANAGER FOR JAVASCRIPT

Hello! My name is Kat Marchán. I'm also @maybekatz of the internet, and I'm the lead maintainer and architect for the npm CLI.

But I'm not here today to just to talk about the npm CLI -- I'm here to talk about a little experiment I've been working on called tink, which I'm hoping will help set a precedent for JavaScript package managers in the future.

Time: 20s

WHAT NEEDS TO HAPPEN?

NPM AS A `node_modules/` BUILDER

But before I get into the what, I want to talk a bit about the *why*. More specifically, I want to talk about what npm needs to do in order to get JavaScript apps to work. And there's really one big end-goal we've had for most of npm's lifetime: put a bunch of stuff into your local `node_modules/` as fast as possible and have everything consume from there.

So that seems straightforward on its own, but what does it really take?

Time: 30s

THE INSTALL PROCESS

OR: WHY THE F--- DID THAT JUST TAKE 10 MINUTES?!

For context, I want to talk about the overall problem, and what steps we've taken along the way to mitigate the issues that came up.

Now the process I'm going to describe is mostly shared between all current JavaScript package managers I know about. They all do different optimizations and clever things around each step, but they all kinda have to do all of this at some point or another in order to be compatible. Let's take a look...

Time: 25s

THE INSTALL PROCESS

1. READ LOCAL DEPENDENCIES (IF ANY)
2. FETCH MISSING PACKAGE METADATA FROM REGISTRY
3. CALCULATE TREE + ACTIONS (FLATTENING SINCE NPM@3)
4. DOWNLOAD + EXTRACT MISSING PACKAGES
5. EXECUTE INSTALL SCRIPTS

So this is kind of a high-level overview of what a JavaScript installer needs to do. We all do this to some extent or another, or we have different optimizations around them, but the concerns are ultimately the same. Let's walk through these steps a bit.

Time: 20s

THE INSTALL PROCESS

1. READ LOCAL DEPENDENCIES (IF ANY)
2. FETCH MISSING PACKAGE METADATA FROM REGISTRY
3. CALCULATE TREE + ACTIONS (FLATTENING SINCE NPM@3)
4. DOWNLOAD + EXTRACT MISSING PACKAGES
5. EXECUTE INSTALL SCRIPTS

First up is this one -- reading local dependencies if you have any. you'll probably notice this one these days when you have a no-op install, that is, when you run `npm` even though you literally just ran it and you're sure all your deps are there.

It usually isn't more than a couple of seconds, but it's still definitely noticeable. Yarn has an interesting optimization around this where they just slap a metadata file inside `node_modules/` based on the hash of `yarn.lock`, and if those match, then they say "fuck it" and trust whatever's in node_modules.

npm has been a bit stubborn here, since we've so far considered the no-op case a fairly rare thing, and if you run npm, we'll be able to auto-fix any issues with your tree, which we think is important enough to warrant the delay. I don't know, we might go ahead and do what Yarn does at some point anyway because people just want that sweet, sweet speedup.

In the end, this step tends to be noticeable, but it's not really the biggest delay.

Time: 60s

THE INSTALL PROCESS

1. READ LOCAL DEPENDENCIES (IF ANY)
2. FETCH MISSING PACKAGE METADATA FROM REGISTRY
3. CALCULATE TREE + ACTIONS (FLATTENING SINCE NPM@3)
4. DOWNLOAD + EXTRACT MISSING PACKAGES
5. EXECUTE INSTALL SCRIPTS

But this next one kinda was, for a while. Turns out making literally thousands of requests just to get metadata from the registry in order to calculate our tree is a fairly network-intensive operation, and in the times before npm@5, this was definitely a time hog.

So what happened? Well, basically, lockfiles happened. Lockfiles are super convenient for developers, sure, but they're really just a single-file cache of the tree calculation step of your package managers, and that's why we love that stuff so much. Once you've built your project up, none of your teammates really need to bother even doing this step until they add or remove a dependency.

This is also why I'm pretty adamant that you should use lockfiles even for libraries. They just make life so much easier.

Time: 60s

THE INSTALL PROCESS

1. READ LOCAL DEPENDENCIES (IF ANY)
2. FETCH MISSING PACKAGE METADATA FROM REGISTRY
3. CALCULATE TREE + ACTIONS (FLATTENING SINCE NPM@3)
4. DOWNLOAD + EXTRACT MISSING PACKAGES
5. EXECUTE INSTALL SCRIPTS

This next one has never really been a huge bottleneck on its own, but I still want to mention it because we do put some effort into this step being fast enough, and I want y'all to know that this step happens to exist.

Long story short, this is what figures out what existing dependencies on disk need to be moved, removed, added, copied, or whatever. Since npm@3, we also do some calculation to flatten that tree, which is fairly fast now.

Anyway, moving on...

Time: 30s

THE INSTALL PROCESS

1. READ LOCAL DEPENDENCIES (IF ANY)
2. FETCH MISSING PACKAGE METADATA FROM REGISTRY
3. CALCULATE TREE + ACTIONS (FLATTENING SINCE NPM@3)
4. DOWNLOAD + EXTRACT MISSING PACKAGES
5. EXECUTE INSTALL SCRIPTS

This one here. This one is by far the heaviest and slowest of all the install steps, mainly because it's the heaviest on pretty much all relevant resources. Let's dive a little deeper into this one.

Time: 15s

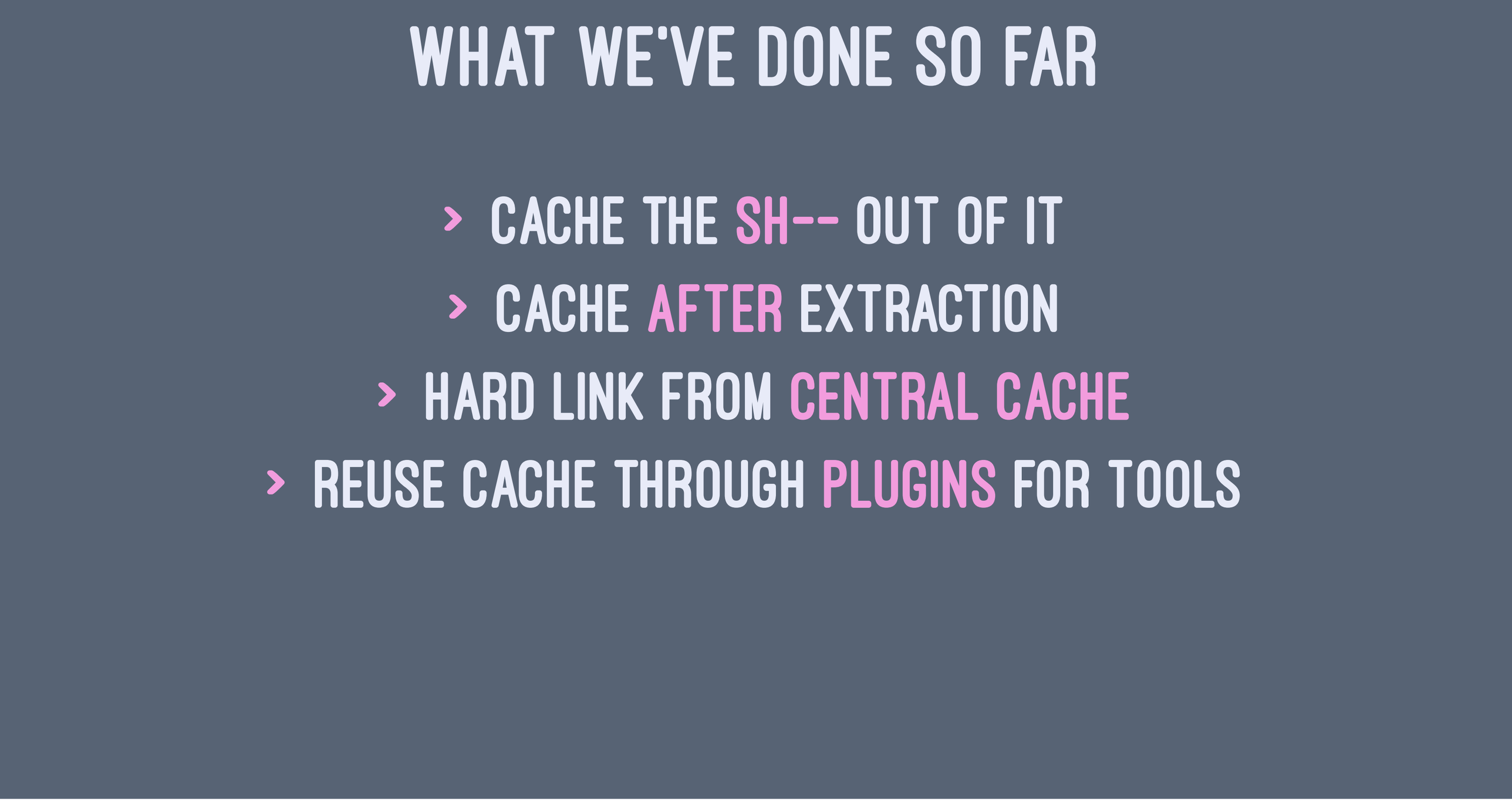
THE EXTRACT STEP

- TARBALL DATA NETWORK TRANSFER (NETWORK)
  - PARSING + UNZIPPING TARBALLS (CPU)
- WRITING FILES TO `node_modules` (DISK I/O, STORAGE)

In the end, we have hundreds of megabytes that need to be transferred over the network, and once we have those, we have to spend a bunch of CPU cycles parsing and gunzipping tarballs, and when *that's* done, we move on to tons of heavy disk I/O. Literally all three of the usual suspects are bottlenecks here one way or another, and it all depends on your usage pattern at some particular point in time.

Long story short this is where all the really bad and slow stuff happens, and this has been a huge target of optimization for pretty much all of us package manager authors in the JavaScript world. And I assume many other package managers, too.

Time: 40s



So what can we do about this? Well, for one, we can do a bunch of caching of tarballs, and different caching strategies have different tradeoffs. npm stores tarballs themselves, which takes up less disk space but it means we have to take the hit of parsing those tarballs every time.

Yarn stores them post-extraction, which uses more space but lets them do very fast copies of all that data instead of re-parsing the tarballs every time.

pnpm is really interesting on this front, where they have an extracted cache, kinda like Yarn, but they hard link all the files into their final destinations. This, surprisingly, isn't much faster than just copying, but it's definitely way more space-efficient because you pay for each package exactly once across your system.

And because I can't seem to stop talking about Yarn today, they have this thing called PnP that partially addresses the problems in this step. The downside with it is that all your tools need to load a plugin, and it mostly works by fussing with the module loader, which can cause some pretty serious incompatibility with the ecosystem. Still, PnP is interesting and definitely worth checking out and I'm sure it's making a lot of lives easier already. You can let Maël tell you about it in the next talk, though.

Time: 80s

THE INSTALL PROCESS

1. READ LOCAL DEPENDENCIES (IF ANY)
2. FETCH MISSING PACKAGE METADATA FROM REGISTRY
3. CALCULATE TREE + ACTIONS (FLATTENING SINCE NPM@3)
4. DOWNLOAD + EXTRACT MISSING PACKAGES
5. EXECUTE INSTALL SCRIPTS

So there's definitely a lot of work going on to make this particular bit better. And, you know, no surprise. It's the most expensive bit. I'll get back to this soon, and what else we can do about it. But for now, let's wrap up this walkthrough. So we do all our downloading and stuff and extracting and then...

Time: 20s

THE INSTALL PROCESS

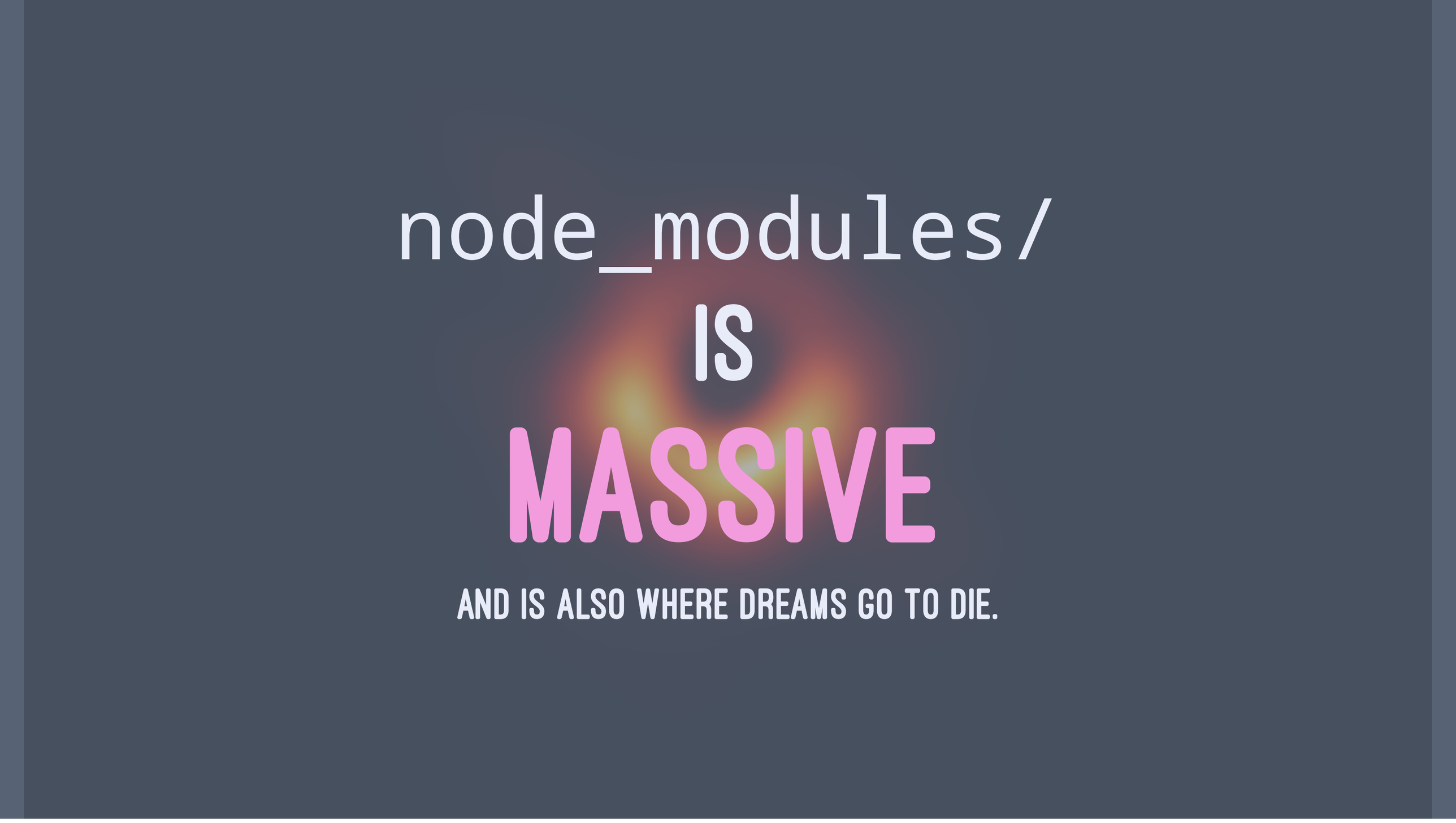
1. READ LOCAL DEPENDENCIES (IF ANY)
2. FETCH MISSING PACKAGE METADATA FROM REGISTRY
3. CALCULATE TREE + ACTIONS (FLATTENING SINCE NPM@3)
4. DOWNLOAD + EXTRACT EXTRACT MISSING PACKAGES
5. EXECUTE INSTALL SCRIPTS

...we have to scan through all the packages we just installed and execute any run-scripts. This isn't usually the biggest step, since there's usually only a couple of run-scripts that run, and most of them aren't that slow, but this still becomes noticeable on some unfortunate projects.

npm doesn't do this, but some other package managers actually parallelize this step, at great risk and complexity, but it does make things just that much faster when you've got heavy stuff to build, like oniguruma. I think we're better off with something like `node-pre-gyp` or a better solution than that, though, to be honest.

What you should take away here, though, is that, in the end...

Time: 40s



The fact is that node_modules/, while a great abstraction, has caused everyone massive headaches in usability and time wasted and disk space abused. There's a lot to agree with on this front with Ryan's talk about Node regrets from last year at this very conference.

Time: 30s

THE FACT REMAINS

- ISOLATED PROJECT DEPENDENCIES ARE GOOD. ACTUALLY.
- BEING SAFE FROM DEPENDENCY HELL IS GOOD. ACTUALLY.
  - WE'RE STUCK WITH A 900K+ PACKAGE ECOSYSTEM.

But what do we actually do about it? It's easy to complain about it and talk about how bad an idea it is, but I still think the good outweighs the bad.

(run through the list)

Time: 30s

SO LET'S DO WHAT WE CAN WITH IT? 

And it's really on us, both the package manager developers, as well as the larger community, to do what we can to make the most of this.

I, of course, have my own ideas about this, and I kinda hinted at what that was, already...

Time: 15s

TINK

MOVING PACKAGE MANAGEMENT INTO THE RUNTIME!

What / think we should do at this point is move package management itself directly into the runtime, instead of an external utility!

And that is what tink is really about, in the end. But what does this mean, and what can we actually do with it, once we've done that?

Time: 20s

```
$ tink sh
```

INSTEAD OF

```
$ node
```

So this is what I'm talking about. You literally stop invoking `node` yourself and you use this new tool called `tink` instead. It has a subcommand called `shell` that wraps node and adds a few patches to make it work its magic.

What magic? well...

Time: 15s

VIRTUAL `node_modules/`

- (MOSTLY) REMOVES PHYSICAL `node_modules/`.
- GLOBALLY DEDUPLICATED FILES BY HASH.
- AUTOMATIC DEPENDENCY FETCHING.
- NO. MORE. `npm install`.

The big idea is that if we control the runtime, we control what happens when anyone tries to read from `node_modules/`. And once we can do that at the runtime level, all sorts of things start happening.

Time: 10s

VIRTUAL `node_modules/`

> (MOSTLY) REMOVES PHYSICAL `node_modules/`. 👉🏼
>> GLOBALLY DEDUPLICATED FILES BY HASH.
>>> AUTOMATIC DEPENDENCY FETCHING.
>>>> NO. MORE. `npm install`.

It means we can get rid of all these copies of `node_modules/` without changing the module loader or the expected APIs from packages. As far as any packages are concerned, they're accessing the filesystem the same way they usually do. And this means it's compatible with little details like `__dirname`, like `fs.readFileSync` to load config files, and literally all of that.

Time: 20s

VIRTUAL `node_modules/`

- (MOSTLY) REMOVES PHYSICAL `node_modules/`.
- GLOBALLY DEDUPLICATED FILES BY HASH. 👈
- AUTOMATIC DEPENDENCY FETCHING.
- NO. MORE. `npm install`.

But if they're not in `node_modules/`, where are the files? Well, instead of copying them, we keep them all in a single global cache, a lot like pnpm does, except we deduplicate every single file at the hash level. That means if you have 5 different versions of the same package, you'll only ever have new copies for the files that actually changed. Storing by hash can also make reads very *very* fast for us.

Time: 30s

VIRTUAL `node_modules/`

- (MOSTLY) REMOVES PHYSICAL `node_modules/`.
  - GLOBALLY DEDUPLICATED FILES BY HASH.
    - AUTOMATIC DEPENDENCY FETCHING. 👉
      - NO. MORE. `npm install`.

But hey, we can do more. Since we control the runtime, why don't we just automatically fetch dependencies for you, as you need them? Why don't we skip downloading dependencies you're not using in your current work?

tink is actually able to block on reads that failed from the local cache and, inline, fetch any packages you haven't downloaded yet. It makes node work a bit more like a browser on that front.

And yes, before you say anything, there's a `--production` flag to disable this if you want to make sure we're not doing random requests for dependencies in production.

Time: 40s

VIRTUAL `node_modules/`

- (MOSTLY) REMOVES PHYSICAL `node_modules/`.
  - GLOBALLY DEDUPLICATED FILES BY HASH.
    - AUTOMATIC DEPENDENCY FETCHING.
      - 😎 NO. MORE. `npm install` 😎

And of course this means you don't do `npm install` anymore. You do `tink add` and `tink rm` to add and remove individual dependencies, and you otherwise don't worry about having an install step anymore. It all just gets done automatically by tink, as needed.

Time: 20s

ISN'T PATCHING `fs` UNSAFE?

Now you might be concerned about all this, by the way.

Patching `fs` itself? Sounds really risky, right?

Time: 5s

ISN'T PATCHING `fs` UNSAFE?

ELECTRON DOES THE SAME THING!

IT WORKS!

But I'm not that worried. This is the same approach Electron has taken and they're fairly successful. We can also learn from their journey so I believe this'll all work out in the end.

Time: 10s

BUT WAIT. THERE'S MORE!

OTHER THINGS TINK ENABLES

- BUILT-IN TYPESCRIPT, ESM, AND JSX SUPPORT
- DEPENDENCY FILE CHECKSUMS AT LOAD TIME!
- TRANSPARENT FETCHING OF MISSING DEPS!
- NO CONFIGURATION!

There's other things that tink allows us to do, now that we have runtime control: tink has typescript, ESM, and JSX support out of the box.

It also runs a very cheap checksum on every single file it loads from the global cache, so you know what you're getting out of the cache is actually the data you want.

If it turns out you ran the tink shell and there's a missing dependency? And if a single file was corrupted in your cache? tink will just download and install it for you. Of course you can turn this behavior off in production, but when you're actually developing, it's super handy.

And finally, my favorite part -- all this comes out of the box without the need to configure or install anything.

Time: 50s

And I want to emphasize this a bit more. All of this comes without the need for any sort of loader. Webpack? It works. Test frameworks like Tap? They'll work too. And all of this without any kind of loader or extra flag or anything. You install and run things with `tink` and that's all you need to do.

Time: 20s

A TOUR OF THE TOOL.

Let's do a brief tour of the tool itself, to give you an idea of what it is I'm actually talking about...

Time: 5s

I WANT TO...

RUN MY NODE APP

```
$ tink sh [file.js]
```

OR: `$ tish [file.js]`

This is the core feature of the entire tool. Just about everything I've talked about so far is available through this one command. Remember that you don't need to do `npm install` anymore. That step is gone. All you do is run `tink sh` and all your dependencies are fetched and extracted as needed by the runtime, as it executes your app. This also works in the interactive shell!

I know I talked about performance a lot before, but a really strong guiding light for me when working on npm has been the idea of simplifying workflows. I want you to have to install and remember as little as possible in order to get productive. The ability to just start running and having the runtime take care of this is super important to me in order to achieve the workflow I think y'all should have in your day-to-day. In the end, I want the package manager to disappear and not be something you have to think about.

Time: 65s

I WANT TO...

EXECUTE A LOCAL DEVDEP/BINARY

$ tink exec <cmd>

Ok so how many of you are familiar with `npx` here? Show of hands?

Anyway, `tink exec` is kinda like `npx`, which for the rest of you is a tool that's bundled with npm itself, and one of the things it does is it lets you execute local bins. For example, if you install `jest` as a devDependency instead of globally, you can use `npx jest` to invoke your devDependency, without needing to install it globally. You don't even need to configure a `run-script` for it.

Running your local bins through `tink exec` means we can apply the same logic to pre-install anything necessary for your binary, that `tink shell` does for scripts.

Time: 40s

I WANT TO...

INSTALL DEPS BEFORE RUNNING

```
$ tink prepare
```

But what if you don't want to slow down your app with package management stuff? That's where `tink prepare` comes in. This command goes ahead and does all the cache warming beforehand, and runs and installs any binaries and the like for you, so when you run `tink shell`, it just works as fast as if you'd done an `npm install` beforehand. You could argue this is the `npm install` of tink, but it's important to note that it's very much an optional step, and it might not speed things up at all. Or it might make the absolute time passed slower, for example, in CI.

Time: 30s

I WANT TO...

HAVE A node_modules/ ANYWAY?

$ tink unwind

In this case there's a nice command called `unwind` that will actually do a full extraction into `node_modules`, so you can use your editor, non-JS build tools, and pretty much anything in the usual way. This is basically an npm install, but there's a slight difference that I'll go into next.

Time: 20s

I WANT TO...

DEBUG A DEPENDENCY

```
$ tink unwind <dep>
```

So what if you want to debug a -specific- dependency, instead of installing a full tree?

Well, then you can do `tink unwind <dep>` and it'll only unwind that one dependency, plus its dependencies. That will let you patch things however you want, debug it, etcetera.

This is something I want to emphasize -- because of the way tink works, anything inside the physical `node_modules` takes precedence over the virtual version. And, in fact, if you use `fs.writeFile` or a similar API, `tink` will actually create a physical file inside `node_modules` for you, which helps immensely with compatibility.

This command is also done automatically at the individual package level for any dependencies that use install scripts, precisely to preserve compatibility, since a lot of run-scripts involve non-JavaScript tools that need to read the actual files.

Time: 55s

I WANT TO...

ADD/REMOVE PACKAGES

```
› $ tink add <pkg>
› $ tink rm <pkg>
› $ tink update <pkg>
```

So since the closest thing to `npm install` only builds existing dependencies, how do we add and remove dependencies, besides editing the `package.json`?

Enter the classic trio, add, rm, and update.

These do basically what it says on the tin: you give them the packages to add and remove and they do it and write out your new `package.json` and `package-lock.json`.

Oh, and by the way, all three of these become interactive if they take no arguments. That means you'll be able to search interactively for new dependencies and pick them from a menu, ditto for removing and updating.

Time: 35s

I WANT TO...

RUN TYPECHECKS. TESTS. LINTER. ETC

```
$ tink check
```

So the last command I'm gonna introduce is this `tink check` utility. The really cool thing about this is that it's a one-stop shop for all your verification and testing. The most noteworthy thing here is that it's going to run your TypeScript typechecks out of the box, without needing to install TypeScript as a devDependency. If you do install it as a devDep, we'll use that, though. But isn't it nice to just be able to go into a project and typecheck the typescript files out of the box? Lint it? It's so nice.

At this point, some of you might wonder if I've been doing too much Rust and Cargo lately, and I think you'd be right. But this isn't a Rust conference, so let's move on.

Time: 40s

WHAT'S NEXT?

- WRAP UP THE PROTOTYPE
- BUILD AN OPEN, RFC-BASED TEAM
- INTEGRATE INTO NPM@8
- WORK WITH NODE CORE TO INTEGRATE IT

So where do we go from here? Well, first, we have to wrap up the prototype because that's all it really is right now: a proof of concept. That should happen soon enough.

The next step is exciting. We want to build an open, RFC-based team that includes more outside contributors than we've usually had in the npm CLI project. This will allow us to really hash out tink and take it where it needs to go, together. The RFC process will ensure there's a community-based approach to growth and development, and it means *you* can make a difference by participating.

Once that's all set, we're turning tink into npm@8. That means that in the future, when you upgrade node, tink will be available out of the box, just like npm currently is. You can still use npm as you usually do, but if you want to opt-in to this new workflow, the tool will be there for you. It'll also integrate with `npx`, meaning all your npx commands will run that much faster.

Finally, once npm@8 is shipped, it's a matter of integrating it into Node itself, and I look forward to working with the folks over there to make this happen. I know it's a big step to be hacking `fs` itself, so this might require a bit of API work on both ends, but I'm sure we can make it happen.

TKTK split up into multiple slides and re-time

Time: 75s

WHAT'S NEXT?

- WRAP UP THE PROTOTYPE
- BUILD AN OPEN, RFC-BASED TEAM
- INTEGRATE INTO NPM@8
- WORK WITH NODE CORE TO INTEGRATE IT
- ONE MORE THING...

Oh, and there's one more thing.

A NEW REGISTRY API?

UNPKG FOR REGISTRY.NPMJS.ORG

We're planning to release, at some point in the future, a new package fetching API for the main npm registry. This is exciting because, combined with tink or something similar, there's a good number of advantages.

Time: 20s

Just to give you an idea, this is why we think this would be great.

The actual API is fairly small: it's a couple of new endpoints that allow us to list the files available in a package, and to fetch those files individually, by hash. That's it, that's pretty much the idea for the new API.

But what do we get from this? Well, for one, some initial analysis on our end estimates that it could involve up to 40% reduction in data transfer from the registry. That means your installs get done a lot faster. It means that if you're on a connection slower than fiber, you'll have less to wait for. It also means you'll store less stuff in your system. You won't need to download READMEs and CHANGELOGs and all that unless your code actually uses them.

This reduction is mainly achieved by having `tink shell` fetch files lazily. So you only download something when you first try to read it, and then we cache it -- aggressively. This is one of the most exciting reasons to do tink, for me.

Finally, and this one's pretty interesting: doing things like this has the potential to completely change how certain monorepo-based libraries work. For example, lodash and babel.

Time: TKTK

As you probably already know, these libraries involve a lot of sub-tools, and they generate a ton of packages that they publish under their scopes, so folks don't have to depend on the entire thing.

This can cause a lot of problems. For example, users can have issues keeping all the packages in sync, if they depend on each other in some way. On the package publisher side, this is a pretty fragile setup because it makes their publish process super prone to publish errors, and they need more and more tooling to make sure, say, they can resume their publish process.

So all of these tiny packages have to be released at the same time, usually all with the same version, and then you have to deal with the registry being eventually consistent, so not all package versions might be available when your user starts installing them, and there's just chaos everywhere.

Time: TKTK

BETTER PUBLISHING FOR MONOREPOS

› PUBLISH `babel` AS SINGLE PACKAGE

› `import '@babel/foo'` -> `import 'babel/foo'`

› ONLY USED SUBPACKAGES DOWNLOADED

Well, with tink and the new registry API, this wouldn't be necessary anymore! Lodash and Babel would be able to publish a single omnibus package, way more than most users would use, but when you install it with tink, it'll only fetch the subpackages you're actually using from within the bigger package. Everyone wins, and you only have to have one lodash or babel item in your `package.json` instead of 12 different things.

Anyway that's that. Like I said, this isn't in progress yet, but it's something we're planning on doing, because the advantages are really clear to us all-around, and I hope you enjoy it if and when it does happen!

Time: TKTK

IN CONCLUSION

- TINK IS EXCITING! AND ALSO A WORK IN PROGRESS!
- VIRTUAL NODE_MODULES/ BY BECOMING YOUR NEW RUNTIME
- TYPESCRIPT. ESM. JSX SUPPORT OUT OF THE BOX
- WILL BE KNOWN AS NPM@8 WHEN RELEASED
- NEW UNPKG-STYLE REGISTRY API COMING IN THE FUTURE.

Time: 25s

MAKING TINK HAPPEN

JOIN US!

- NPM.COMMUNITY/C/DEVELOPMENT
- GITHUB.COM/NPM/RFCS
- @MAYBEKATZ ON TWITTER
- @ZKAT ON GITHUB