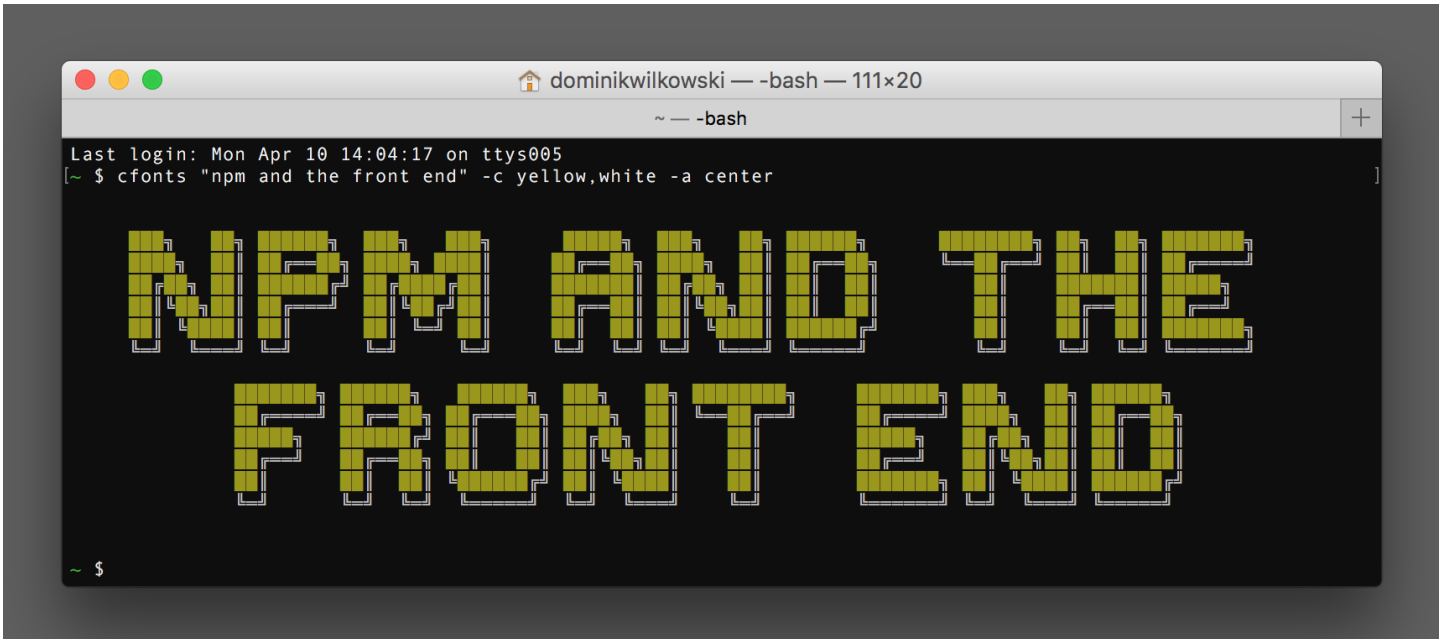
Dominik
Wilkowski
Design systems developer. Mostly NodeJs work. Loves sweets. Also movie-buff.
Apr 9, 2017 · 8 min read



npm and the front end

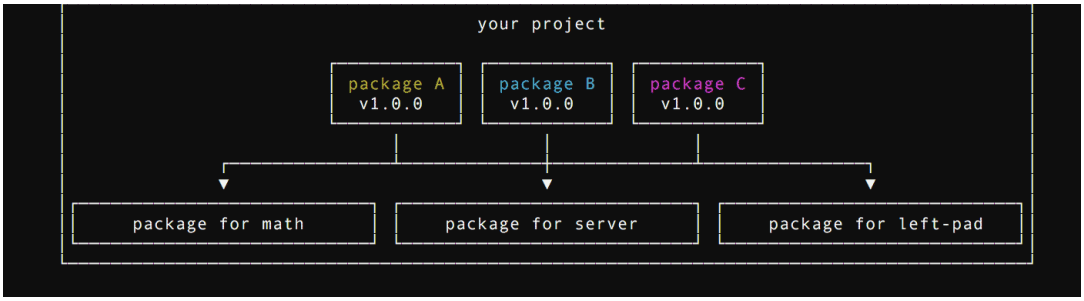
or the virtue of breakfast metaphors

Working with npm on the front end comes with many issues. A summary of this can be found in this npm blog post. In the following article, I explore the way we use and reuse front end components (such as Sass partials) seamlessly through the npm registry.

The problem

For the purpose of this article let's look at the most fundamental issue for the front end: dependency hell.

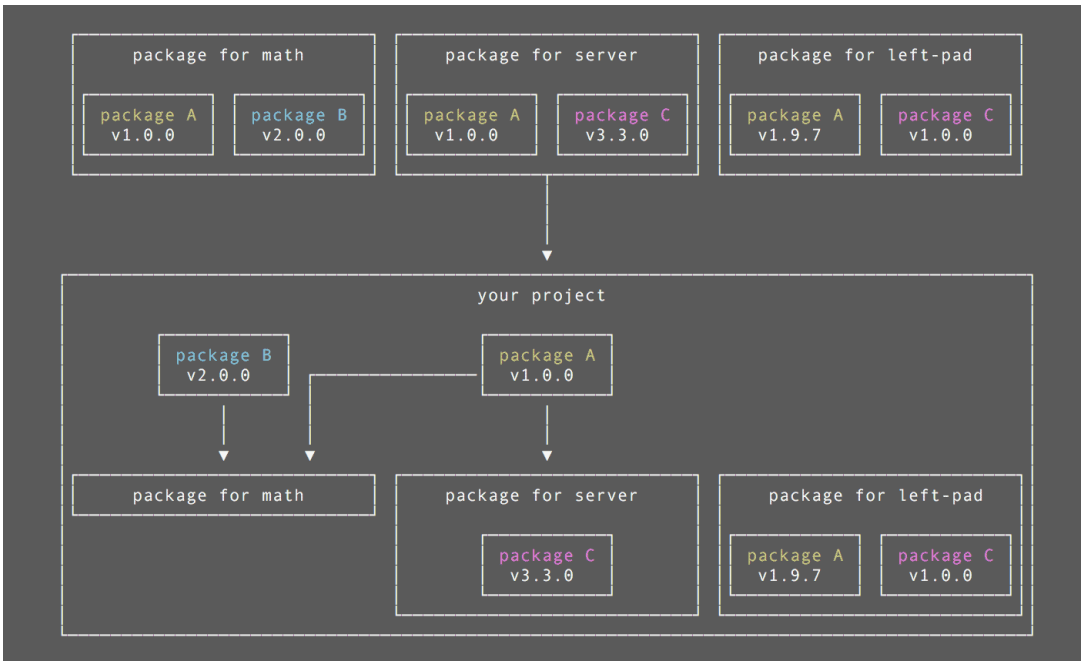
Npm embraces modular architecture of software development. Do one thing only and do it well. This works well in theory and means you can pick from a gazillion great modules to build your thing.



Npms perfect use case where all dependencies can be reused without conflicts

The example above shows how this would work in a perfect world. Each module can reuse all of it's dependencies. We don't double up on code and all together our project ends up with **3 dependencies**.

The reality, however is different.



A more realistic view of how dependencies are managed via npm

Packages will come with dependencies that conflict with other package dependencies. In this, still very reduced example we end up with **5 dependencies** for our project. That's because even though each package still requires the same dependencies, they do so on different versions. Npm makes sure everything still works by installing package A and B twice under the packages that required them.

Now imagine each dependency has it's own dependencies and most of them conflict as well. It's not uncommon in the npm ecosystem to end up with 30,000 or more dependencies.

This is not an issue when you use npm for the back end. There, we don't need to care about the size of our project. On the front end however, where each kilobyte counts and each millisecond is optimised we have no affordance. Now think about CSS-land where each variable or declaration is global. Even with really smart JavaScript workarounds we still end up doubling up on things that should naturally cascade.

Many things have been written about this topic and most involve putting CSS into

Javascript. I think that is a very good solution for projects that entirely rely on Javascript. In others, where we want don't want to introduce a large Javascript framework to manage and scope our CSS, it might not be the right way to go about it. I think sharing CSS, Sass, postCSS etc. should not depend on your Javascript flavour as CSS is one of the main three languages the web understands natively after all.

Where we should go

I want to lock step with npm and build CSS packages for the front end. Modular software architecture has been around for a while and it's time for us front end developers to get with it. Building your software from a lot of tiny pieces that have been individually tested is better bang for your buck.

Ideally you should be able to build small front end components and share them in the registry. No need to reinvent buttons every time you start a new project. Making them adjustable to suit your needs is easy with CSS/Sass variables. Sass partials are a common way to build your site in a modular way. We just don't really reuse and distribute them yet. We also don't version each of them. It would make things more complex and needs endless tooling, right? But why can't we use what npm has already built? A huge registry of reusable, versioned bits that can be easily installed and reused.

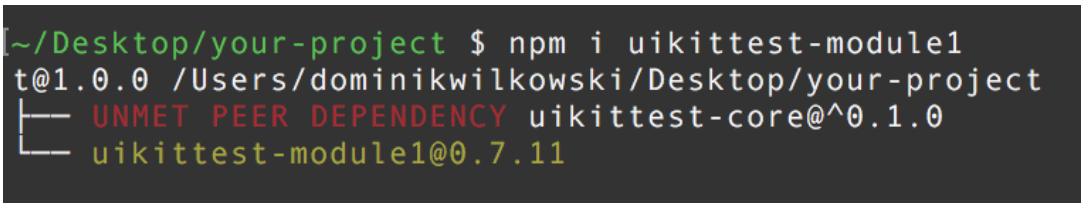
I'm working on a design system for the Australian Government and have had to solve this problem. We needed a way to easily distribute a library of UI components. Maybe, just maybe, it will also help you start using npm packages in your front end.

The solution

When looking at the problem in detail I came up with three goals I had to reach for this to be successful:

1. Keep dependencies flat and error out conflicts
2. Use existing technology that is reasonably well known
3. Don't force tooling, environment or preferences

I decided to use peer dependencies, something npm has built in, but does not enforce anymore since npm 3. Peer dependencies mean the required package needs to be on the same nesting level as the root package.

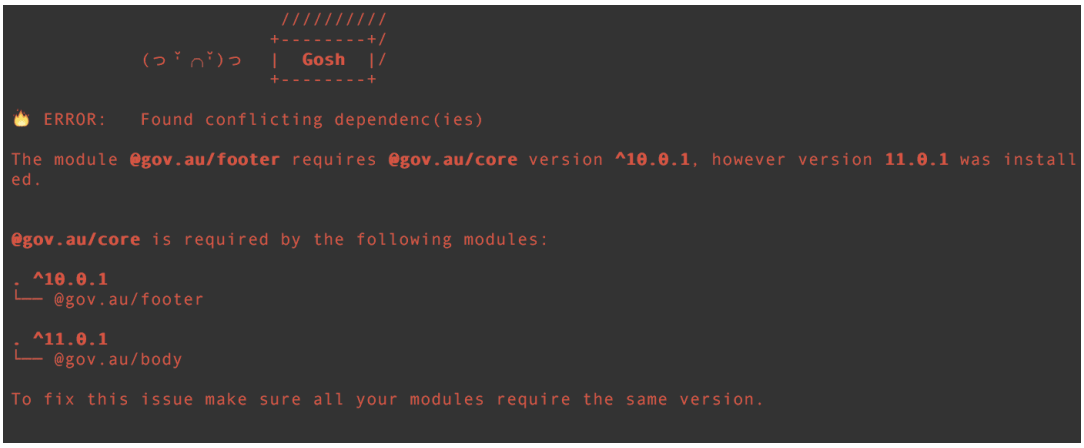


The warning npm throws when a package with peer dependencies is installed

If you have a package with peer dependencies, all npm will do is give you the warning above. It won't install the dependency automatically for you nor will it even error out. Just a warning. Although peer dependencies are the appropriate paradigm, the way they are handled needed enhancing. How do we make sure all dependencies are flat?

Enter Pancake,

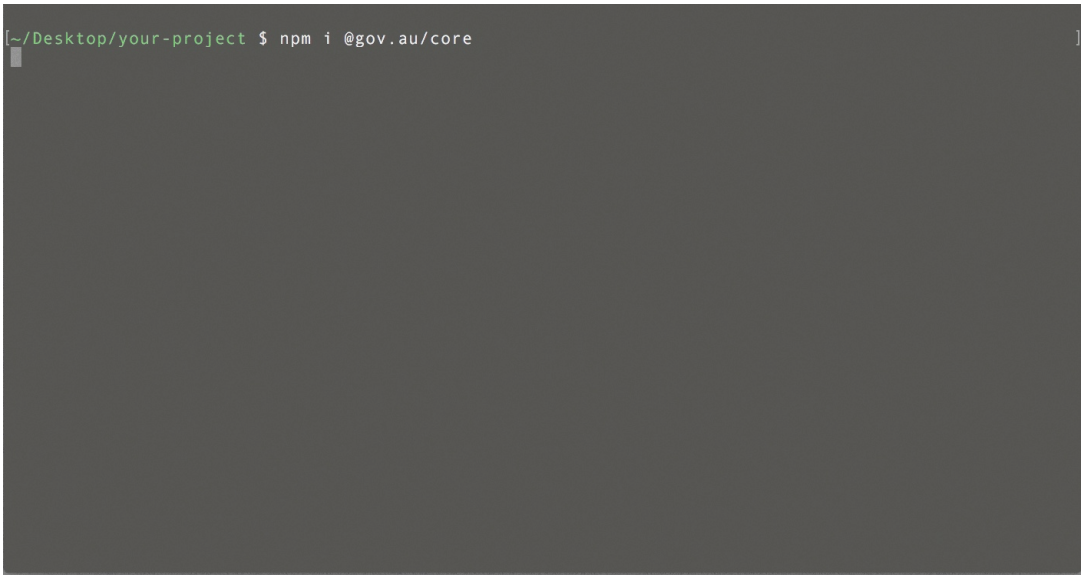
the tool I created to handle the error messaging around peer dependencies and then some. We called it Pancake because it makes things flat. Really, that's where the metaphor ends. Pancake will check all your dependencies for peer dependency conflicts and error out with a helpful error message.



This is what a conflict with Pancake looks like

But how do we introduce pancake into your toolchain without making you adopt, learn and use yet another thing?

I opted for the silent approach: by adding pancake as a dependency to each flat package and run it from the post install hook of those packages. That means you don't have to install it or even run it manually. It will just run every time you install a package that has been prepared for Pancake.



Pancake runs right inside the npm install command

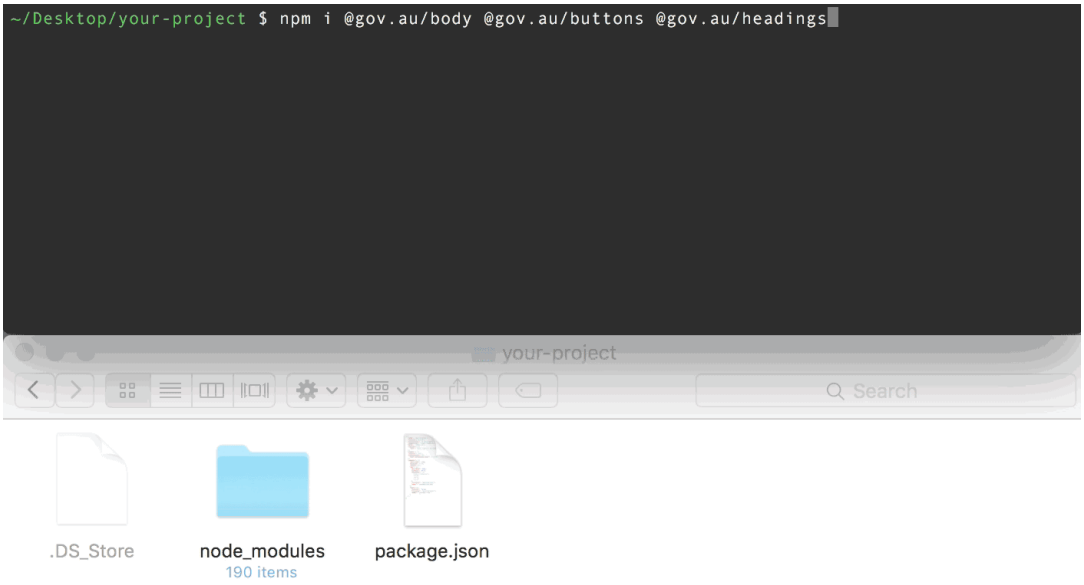
We now have certainty that packages you install are flat. No more dependency hell. That's a start but it leaves us with having to import each package by hand into your sass files.

```
@import '../../node_modules/your-packageA/dist/_index.scss';
```

```
@import '../../node_modules/your-packageB/path/to/_index.scss';
@import '../../node_modules/your-packageC/_packageC.scss';
```

That's a lot of work if you want to really reuse small components quickly. Since Pancake runs already, I added a plugin system to it. Pancake is installed, downloaded and executed after each package is installed, so I thought it to be important to make it as lightweight as possible. I abstracted all extra functionality into plugins. The main application comes with only one dependency that is needed to get it to run in node 5 and up. After checking your peer dependencies for conflicts, Pancake will then look into your `package.json` file for its configuration. If you haven't opted out of plugins, Pancake will install and run plugins for you that handle Sass, postCSS or whatever else you might deliver in your flat packages. Those plugins can compile and save files for you each time you install a new package and put it into a place of your choosing. A plugin e.g. could compile your Sass so you don't have to commit CSS code. We have a core package that comes with a lot of variables. Depending on what version of the core you use, the CSS of your package might differ making it impossible to precompile it. You might also want an SVG plugin. It would generate PNGS fallbacks from your SVGs and create a SVG spritesheet.

How will this sit in your workflow?



Installing a Pancake module will generate the import file for you

The **pancake-sass** plugin will generate and save a file with all current Pancake packages already included. All you have to do is include that file into your Sass project and each time you install a new package, your project will automatically update. This means you get the new Sass code right away. No need to search for the new path inside your `node_modules` folder and import it by hand.

The files Pancake generate are artefacts of your `package.json` file and can be happily ignored by version control systems like git. Whatever you do, the same `package.json` file will always generate the very same files in the very same folders. *(This is different to the bower approach where conflicts would give you a choice, so one `package.json` file could*

generate wildly different outcomes)

Whoa

There are a lot of other things Pancake can do for you. It introduces a toolchain you can extend yourself for all flat packages. They don't have to be CSS only. Pancake makes sure all settings are saved into your `package.json` file. That way everyone is on the same page. Of course you can disable that too. Head over to the GitHub Readme to learn more.

But what about my ci builds?

We work with continuous integration ourself and know the importance of fast ci builds. That's why the behaviour of Pancake is configurable. You can change the folder in which you want to save your import files, ignore certain plugins or switch them off entirely. *Beware though, some ci services rely heavily on caching and won't run the actual `npm install` command. In that case you'll have to instruct your build to run pancake manually after `npm install` .*

How can I try this?

Head over to the Australian GOV.AU UI-Kit to try it out. Each module of the Design System is delivered with Pancake and it is super fast to use and reuse each module.

Just create a new folder with a `package.json` file *(Pancake needs the `package.json` file or it will fail like a messy, oil spitting kitchen accident)* and start installing packages:

```
npm install @gov.au/header @gov.au/body @gov.au/footer
```

Now include the generated `pancake.sass` file into your main Sass file. Whenever you feel like you're missing a package, just install it and your Sass files will update automatically.

```
npm install @gov.au/buttons
```

This has been hugely non-disruptive to my workflow. It works super fast and I don't have to think about Sass partials again. I can share them much quicker and get to the hard bits faster.

No more reinventing the wheel.

JavaScript NPM Sass Atomic Design Design System

Dominik Wilkowski

Design systems developer. Mostly NodeJs work. Loves sweets. Also movie-buff.

Follow

DailyJS

JavaScript news and opinion.

Follow

Never miss a story from **DailyJS**

GET UPDATES