The bloat of edge-case first libraries

September 09, 2025 Edit Page

This is just some of what I've been pondering recently - particularly in terms of how we ended up with such overly-granular dependency trees.

I think we've ended up with many libraries in our ecosystem which are edge-case-first, the opposite to what I'd expect. I'll give a few examples and some thoughts around this, mostly in the hope we can start to trim some of it away.

The problem

I believe a lot of the questionably small libraries hiding in our deep dependency trees are a result of over-engineering for inputs and edge cases we've probably never seen.

For example, say we're building a `clamp` function:

```
export function clamp(value: number, min: number, max: number): number {
  return Math.min(Math.max(value, min), max);
}
```

Pretty simple!

What if someone passes nonsensical ranges? Let's handle that.

```
export function clamp(value: number, min: number, max: number): number {
  if (min > max) {
    throw new Error('min must be less than or equal to max');
  }
  return Math.min(Math.max(value, min), max);
}
```

This is probably as far as I'd go. But let's over-engineer - what if someone passes a number-like string?

```
export function clamp(value: number | string, min: number | string, max: number
  if (typeof value === 'string' && Number.isNaN(Number(value))) {
    throw new Error('value must be a number or a number-like string');
  }
  if (typeof min === 'string' && Number.isNaN(Number(min))) {
    throw new Error('min must be a number or a number-like string');
  }
  if (typeof max === 'string' && Number.isNaN(Number(max))) {
    throw new Error('max must be a number or a number-like string');
  }
  if (Number(min) > Number(max)) {
    throw new Error('min must be less than or equal to max');
  }
  return Math.min(Math.max(value, min), max);
}
```

At this point, it seems clear to me we've just poorly designed our function. It solely exists to clamp numbers, so why would we accept strings?

But hey, let's go further! What if other libraries also want to accept such loose inputs? Let's extract this into a separate library:

```
import isNumber from 'is-number';

export function clamp(value: number | string, min: number | string, max: number
  if (!isNumber(value)) {
    throw new Error('value must be a number or a number-like string');
  }
  if (!isNumber(min)) {
    throw new Error('min must be a number or a number-like string');
  }
  if (!isNumber(max)) {
    throw new Error('max must be a number or a number-like string');
  }
  if (Number(min) > Number(max)) {
    throw new Error('min must be less than or equal to max');
  }
  return Math.min(Math.max(value, min), max);
}
```

Whoops! We've just created the infamous `is-number` library!

How it should be

This, in my opinion, is poor technical design we've all ended up dealing with over the years. Carrying the baggage of these overly-granular libraries that exist to handle edge cases we've probably never encountered.

I think it should have been:

```
export function clamp(value: number, min: number, max: number): number {
  return Math.min(Math.max(value, min), max);
}
```

Maybe with some `min <= max` validation, but even that is debatable. At this point, you may as well inline the `Math.min(Math.max(...))` expression instead of using a dependency.

We should be able to define our functions to accept the inputs they are designed for, and not try to handle every possible edge case.

There are two things at play here:

- Data types
- Values

A well designed library would assume the right **data types** have been passed in, but may validate that the **values** make sense (e.g. `min` is less than or equal to `max`).

These over-engineered libraries have decided to implement *both* at runtime - essentially run-time type checking and value validation. One could argue that this is just a result of building in the pre-TypeScript era, but that still doesn't justify the overly specific *value validation* (e.g. the real `is-number` also checks that it is finite).

What we shouldn't do

We shouldn't build edge-case-first libraries, i.e. those which solve for edge cases we have yet to encounter or are unlikely to ever encounter.

Example: `is-arrayish` (76M downloads/week)

The `is-arrayish` library determines if a value is an `Array` or behaves like one.

There will be some edge cases where this matters a lot, where we want to accept something we can index into but don't care if it is a real `Array` or not.

However, the common use case clearly will not be that and we could've just used `Array.isArray()` all along.

Example: `is-number` (90M downloads/week)

The `is-number` library determines if a value is a positive, finite number or number-like string (maybe we should name it `is-positive-finite-number` to be more accurate).

Again, there will be edge cases where we want to deal with number-like strings or we want to validate that a number is within a range (e.g. finite).

The common use case will not be this. The common use case will be that we want to check `typeof n === 'number'` and be done with it.

For those edge cases where we want to *additionally* validate what kind of number it is, we could use a library (but one which exists for the validation, not for the type check).

Example: `pascalcase` (9.7M downloads/week)

The `pascalcase` library transforms text to PascalCase.

It has 1 dependency (`camelcase`) and accepts a variety of input types:

- strings
- null
- undefined
- arrays of strings

- functions
- arbitrary objects with `toString` methods

In reality, almost every user will be passing a `string`.

Example: `is-regexp` (10M downloads/week)

The `is-regexp` library checks if a value is a `RegExp` object, and supports cross-realm values.

In reality, almost every user will be passing a `RegExp` object, and not one from another realm.

For context, cross-realm values can happen when you retrieve a value from an `iframe` or VM for example:

```
const iframe = document.createElement('iframe');
iframe.contentWindow.RegExp === RegExp; // false

const iframeRegex = iframe.contentWindow.someRegexp;

iframeRegex instanceof RegExp; // false
isRegex(iframeRegex); // true
```

This is indeed useful, and I do support this myself in chai (which I maintain). However, this is an edge case most libraries don't need to care about.

What we should do

We should build libraries which solve the common use case and make assumptions about the input types they will be given.

Example: scule (1.8M downloads/week)

scule is a library for transforming casing of text (e.g. camel case, etc).

It only accepts inputs it is designed for (strings and arrays of strings) and has zero dependencies.

In most of the functions it exports, it assumes valid input data types.

Example: dlv (14.9M downloads/week)

dlv is a library for deep property access.

It only accepts strings and arrays of strings as the path to access, and assumes this (i.e. does no validation).

Validation is important

Validation is important, and I want to be clear that I'm not saying we should stop validating our data.

However, we should usually be validating the data in the project that owns it (e.g. at the app level), and not in every library that later consumes it as input.

Deep dependencies applying validation like this actually shift the burden from where it belongs (at data boundaries) to deep in the dependency tree.

Often at this point, it is invisible to the consumer of the library.

How many people are passing values into `is-number` (via other libraries), not realising it will prevent them from using negative numbers and `Infinity`?

A note on overly-granular libraries

This post isn't about overly-granular libraries in general, but I'd like to briefly mention them for visibility.

An overly-granular library is one where someone took a useful library and split it up into an almost atomic-level of granularity.

Some examples:

- `shebang-regex` - 2LOC, does the same as `startsWith('#!')`, **86M downloads/week**
- `is-whitespace` - 7LOC, checks if a string is only whitespace, **1M downloads/week**
- `is-npm` - 8LOC, checks `npm_config_user_agent` or `npm_package_json` are set, **7M downloads/week**

This is a personal preference some maintainers clearly prefer. The thought seems to be that by having atomic libraries, you can easily build your next library mostly from the existing building blocks you have.

I don't really agree with this and think downloading a package for `#!` 86 million times a week is a bit much.

What can be done about this?

The e18e community is already tackling a lot of this by contributing performance improvements across the ecosystem, including removing and replacing dependencies with more modern, performant ones.

Through these efforts, there's already a useful list of replacements and an ESLint plugin.

As a maintainer

If you're maintaining a library, it would be worth reviewing your dependencies to see if:

- Any are replaceable by native functionality these days (e.g. `Array.isArray`)
- Any are replaceable by smaller, less granular and/or more performant alternatives (e.g. `scule` instead of `pascalcase`)
- Any are redundant if you make more assumptions about input types

Tools like npmgraph can help you visualise your dependency tree to make this task easier.

Also, being stricter around input types will allow you to reduce a lot of code and dependencies.

If you can assume the data being passed in is the correct type, you can leave validation up to the consumer.

As a user

Keep a close eye on your dependencies (both deep and direct), and what alternatives are available to your direct dependencies.

Often, it is easy to stick with a dependency from long ago and forget to re-visit it one day in case there is a better way. Many of these packages are possible natively, or have more modern alternatives.

Useful tools:

- npmgraph for visualising your dependency tree
- node-modules.dev for visualising your dependencies and lots of useful meta data
- Dependabot for keeping your dependencies up to date

On the topic of data, it is also worth ensuring validation happens at data boundaries rather than being delegated to various dependencies. Try to validate the type and value up front, before passing into dependencies.

Conclusion

Most of these libraries exist to handle edge cases that do certainly exist. However, **we are all paying the cost of that rather than only those who need to support those edge cases**.

This is the wrong way around. Libraries should implement the main use case, and alternatives (or plugins) can exist to provide the edge cases the minority needs.

We should all be more aware of what is in our dependency tree, and should push for more concise, lighter libraries.