dont-break - Check if you break dependents - Interview with Gleb Bahmutov

17 Jul 2017

Releasing new versions of npm modules is an `npm publish` away. But how do you make sure you don't accidentally break a dependent project? Even if you are careful and test your module well, someone may be depending on a behavior you are not testing.

dont-break, a tool by Gleb Bahmutov, was designed to address this problem.

Can you tell a bit about yourself?

Gleb Bahmutov

I have a degree in computer vision. I did panorama stitching and 3D scanners for a while which meant heavy duty C++ programming. At a certain point, I wanted to expand the areas of the business that I worked in: beyond image acquisition and processing. I also wanted to show results and make them useful to the users, for which I needed JavaScript. Now I mostly do Node.js and JS hacks.

How would you describe dont-break to someone who has never heard of it?

Here is the problem with software development today: there are tl registry: Maven for Java, npm for JavaScript, and so on. Each module has multiple versions. Your module probably depends on some modules (upstream dependencies) and other modules may depend on your module (downstream dependencies).

Updating Upstream Dependencies

If there are new versions of upstream dependencies, you should probably upgrade versions carefully. I wrote next-update, a CLI tool that tries each new version of an upstream dependency, runs your tests and if the tests pass, upgrades the dependencies to their new versions.

There are services that automate this, like GreenKeeper and renovate. The feedback loop is pretty quick: the tool installs a new version of an upstream dependency, runs your tests and displays the result given the new version is compatible or breaks your module.

Editor's note: Read the interview about renovate to learn more about it.

The Problem of Downstream Dependencies

However, this does not address issues in downstream dependencies caused by new versions of your module. Maybe you changed the API or released a bug with the new version, and the downstream dependencies cannot upgrade without breaking their tests.

The feedback loop is super long - you publish a new version, a while later maintainers of a downstream dependency try to upgrade to the new version, their tests fail, and finally a bug is opened in your project (if they feel generous).

dont-break turns the tables - you can test downstream dependencies with new, unpublished versions of your module to see if the new code breaks them or not.

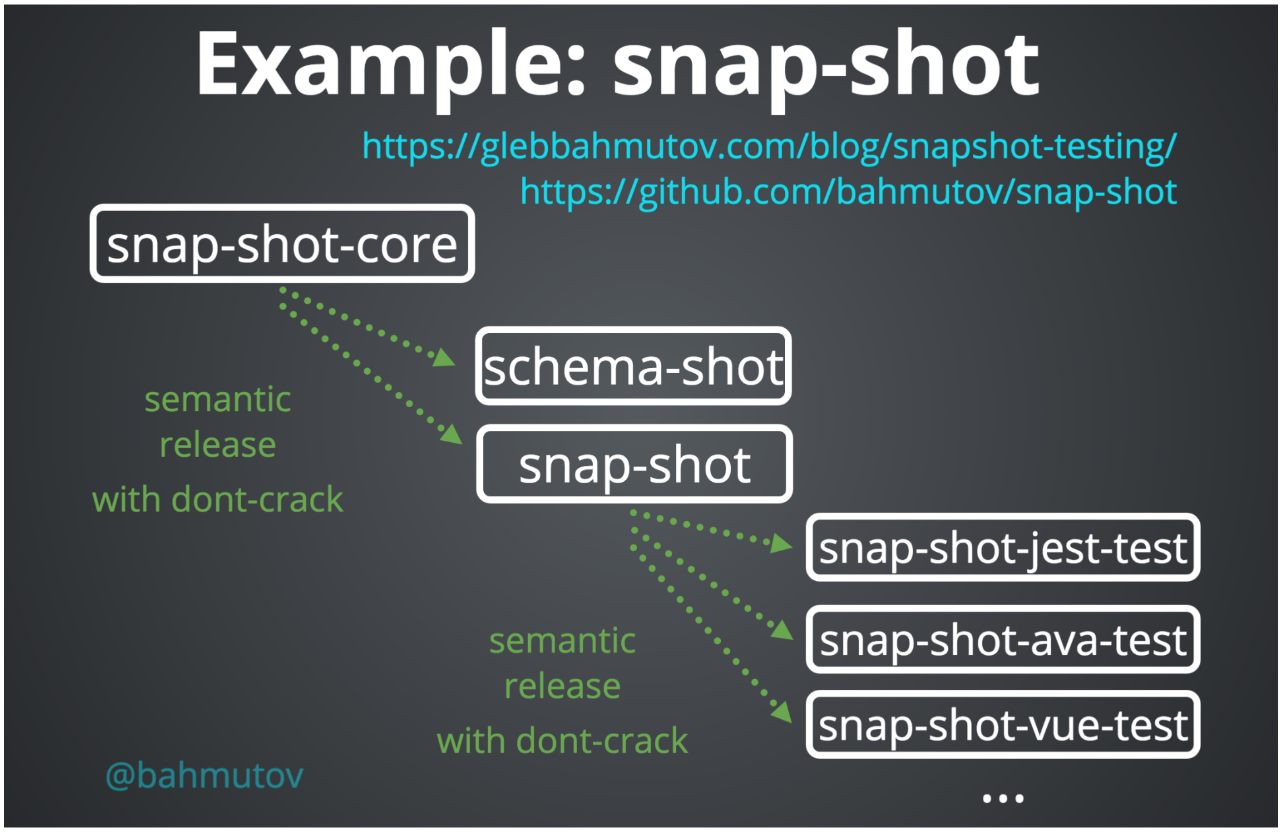
How does dont-break work?

dont-break is a CLI tool that can find and test downstream dependencies of your npm module. Here's the basic algorithm:

1. For each downstream dependency, firstly the repository will be found and cloned.
2. To ensure the dependency's tests run from a new clone, the tests will be run once using *the dependency's version of your module*.
3. If they pass, the new, unpublished version of your module will be copied into the dependency's `node_modules` directory, and the tests run again.
4. If the tests also pass with your unpublished version, this dependency can be considered functional with your new version.

This can be done for as many downstream dependencies as you'd like, and if no relevant issues are found, the new version of your module can be safely published.

A good example is snap-shot-core, which is checked against its downstream dependencies *snap-shot* and *schema-shot*, among others. The project *snap-shot* is in turn checked against its downstream dependencies *snap-shot-jest-test*, *snap-shot-ava-test*, etc. In diagram form it looks like this:


Example: snap-shot
https://glebbahmutov.com/blog/snapshot-testing/
https://github.com/bahmutov/snap-shot
snap-shot-core
schema-shot
snap-shot
semantic release with dont-crack
snap-shot-jest-test
snap-shot-ava-test
snap-shot-vue-test
semantic release with dont-crack
...
@bahmutov

snap-shot in a diagram form

The above slide is from my presentation that I highly recommend to anyone working in the npm ecosystem: Self Improving Software. It demonstrates *dont-break* via the *dont-crack* wrapper (see below).

How does dont-break differ from other solutions?

As far as I know, there are no similar tools. Few people like publishing new versions of their modules that are compatible with previously published ones as much as I do (just kidding). Hopefully, in the near future, each project will always stay up to date and will be carefully tested against existing "users" (downstream dependencies) before releasing a patch version for example.

Why did you develop dont-break?

I love, love, love writing new software (you can find links to my op

Table of contents

easiest way to produce a lot of useful software is to do two things:

1. Use existing modules rather than writing your own
2. Keep dependencies up to date to benefit from new features and bug fixes written by others

After publishing some modules, I noticed that I would routinely set up a lot of the same tools for each of them, so I automated my project setup. By using semantic-release, I also achieved an automated publishing process.

The problem was that I still had to write tests to make sure minor and patch releases didn't cause issues for existing "users" (downstream dependencies). To avoid extra testing work, I started cloning downstream projects, copying my new code into those folders and running the tests. Bingo! The idea was born: why don't I automate this?

What next?

I have two things planned for the future (well, one is well under way, the other is still just a vague idea).

dont-crack

There is a *semantic-release* plugin I wrote called dont-crack that wraps *dont-break* to avoid having to run it locally. My Continuous Integration setup looks at the commits since the last release, and if it decides that it should publish a new `minor` or `patch` version (meaning, there should be no breaking API changes), it runs *dont-break* to confirm that downstream dependencies do not break.

If a downstream dependency does break, that means our change was *incompatible*, and it should be published as a *breaking major* change. Doing this lets the downstream maintainers know there is an update, but it will require some work. Otherwise, we are all good and can safely publish a `minor` or `patc`

Table of contents

Sending Reports Back to Other Projects

I am interested in how to send good bug reports back to your module if one of the downstream dependencies suddenly breaks. Imagine a new version of your module is breaking someone else's module. Do I just see a stack trace from their module - a project I do not maintain or know? Or can we somehow show precisely *what behaves differently* between the previous version of your module and the latest code?

Solving the second problem will finally enable large source code monorepos to be split up. Working on an individual component and being able to test it against its dependencies in a nice, fast and useful way would be huge!

What trends do you see in the future? What advice would you give to someone starting with web development?

I see a huge push towards immutable *everything*. I see this in data structures (you cannot modify an object because someone else relies on it), deployment (Docker, immutable infrastructure) and the npm registry (cannot unpublish a version already there).

The concept of "this artifact is permanent and is not going to disappear" is nice. Thus my advice would be to learn how to update objects without mutating the original ones, find out how to deploy many times per day and learn about deploying a new system instead of *tinkering with a running system*.

Who should I interview next?

Please, please interview my coworker, Brian Mann the founder of Cypress.io. Of course, it is a shameless plug, since we work together on the fantastic E2E testing tool (it is going to be open sourced soon, I promise). He has a good sense of what makes web application feature testing hard and why existing

Selenium are not enough. Also, I disagree with Brian a lot, but love hearing his take on things!

Conclusion

Thanks for the interview Gleb! I think if the community adopted tools like *dont-break*, that would be one step forward in solving the npm package quality problem. Perhaps some of these problems could be pushed to a service level to help people improve the quality of their work while enhancing the ecosystem.

To learn more about *dont-break*, check out the GitHub project.

17 Jul 2017

Juho Vepsäläinen

Subscribe to the blog updates

Previous post

← Rollup - Next-generation ES6 module bundler - Interview with Rich Harris

Next post

Fall Tour - Vienna Clinics, ReactNext, WebExpo, ReactiveConf →

Table of contents