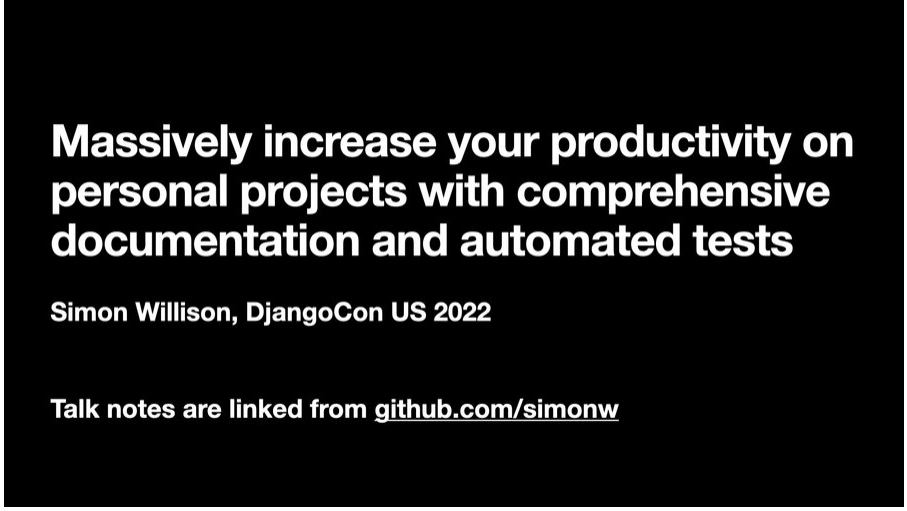
Massively increase your productivity on personal projects with comprehensive documentation and automated tests

I gave a talk at DjangoCon US 2022 in San Diego last month about productivity on personal projects.

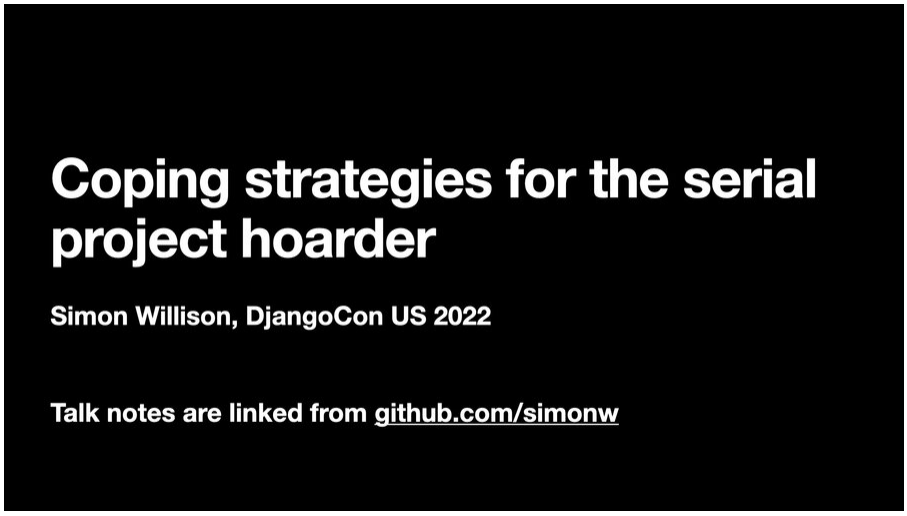
I’m maintaining a *lot* of different projects at the moment. Somewhat unintuitively, the way I’m handling this is by scaling down techniques that I’ve seen working for large engineering teams spread out across multiple continents.

The key trick is to ensure that every project has comprehensive documentation and automated tests. This scales my productivity horizontally, by freeing me up from needing to remember all of the details of all of the different projects I’m working on at the same time.

You can watch the talk on YouTube (25 minutes). Alternatively, I’ve included a detailed annotated version of the slides and notes below.

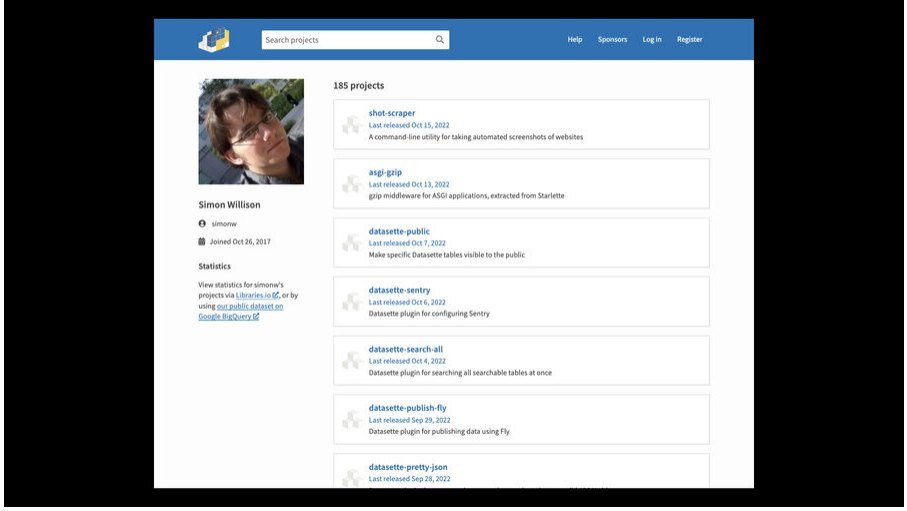


This was the title I originally submitted to the conference. But I realized a better title was probably...



Coping strategies for the serial project hoarder

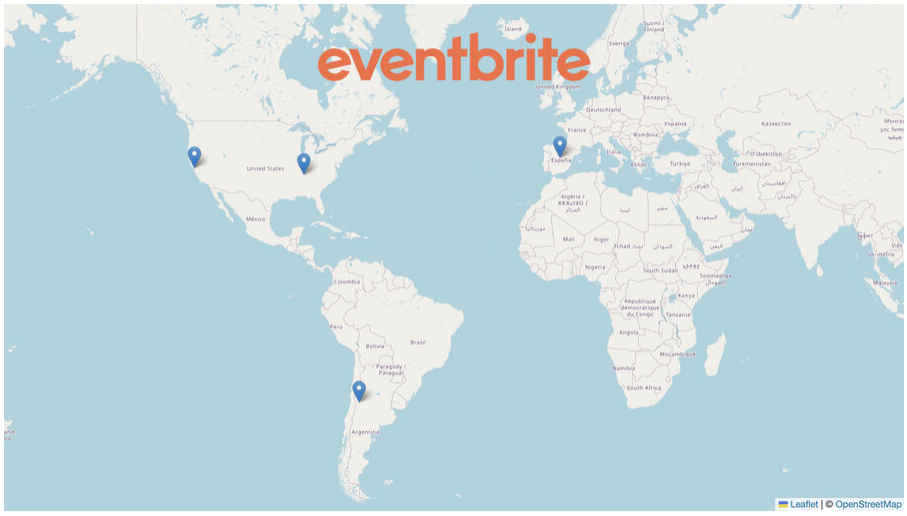
This video is a neat representation of my approach to personal projects: I always have a few on the go, but I can never resist the temptation to add even more.



My PyPI profile (which is only five years old) lists 185 Python packages that I've released. Technically I'm actively maintaining all of them, in that if someone reports a bug I'll push out a fix. Many of them receive new releases at least once a year.

Aside: I took this screenshot using shot-scraper with a little bit of extra JavaScript to hide a notification bar at the top of the page:

```
shot-scraper 'https://pypi.org/user/simonw/' \
--javascript "
    document.body.style.paddingTop = 0;
    document.querySelector(
        '#sticky-notifications'
    ).style.display = 'none';
  " --height 1000
```



How can one individual maintain 185 projects?

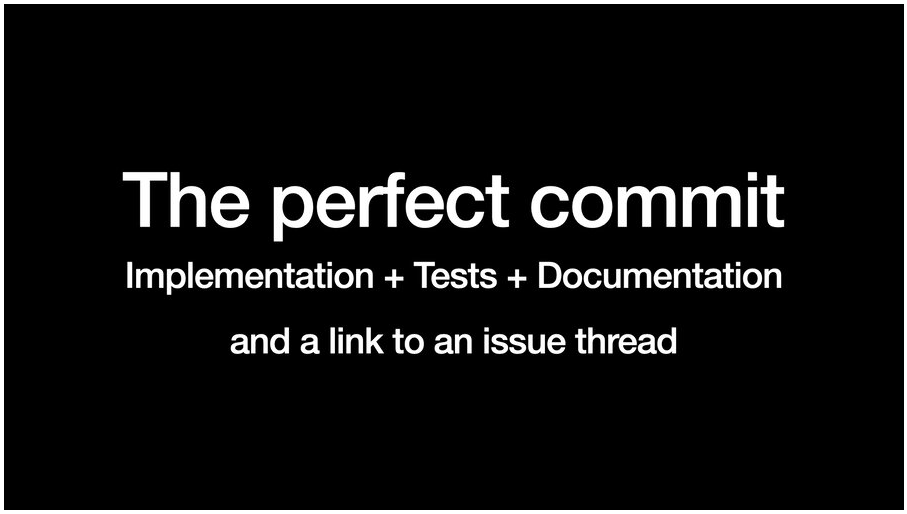
Surprisingly, I'm using techniques that I've scaled down from working at a company with hundreds of engineers.

I spent seven years at Eventbrite, during which time the engineering team grew to span three different continents. We had major engineering centers in San Francisco, Nashville, Mendoza in Argentina and Madrid in Spain.

Consider timezones: engineers in Madrid and engineers in San Francisco had almost no overlap in their working hours. Good asynchronous communication was essential.

Over time, I noticed that the teams that were most effective at this scale were the teams that had a strong culture of documentation and automated testing.

As I started to work on my own array of smaller personal projects, I found that the same discipline that worked for large teams somehow sped me up, when intuitively I would have expected it to slow me down.

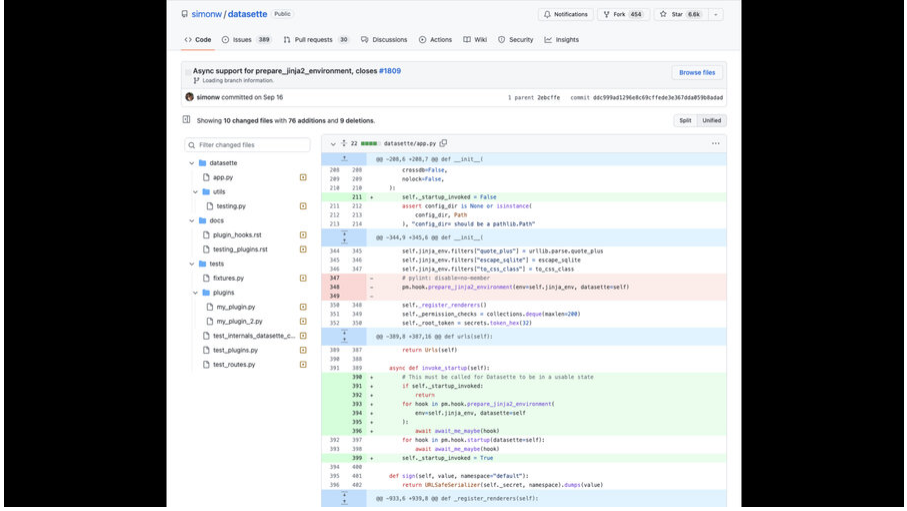


I wrote an extended description of this in The Perfect Commit.

I've started structuring the majority of my work in terms of what I think of as "the perfect commit"—a commit that combines implementation, tests, documentation and a link to an issue thread.

As software engineers, it's important to note that our job generally isn't to write new software: it's to make changes to existing software.

As such, the commit is our unit of work. It's worth us paying attention to how we cen make our commits as useful as possible.



Here's a recent example from one of my projects, Datasette.

It's a single commit which bundles together the implementation, some related documentation improvements and the tests that show it works. And it links back to an issue thread from the commit message.

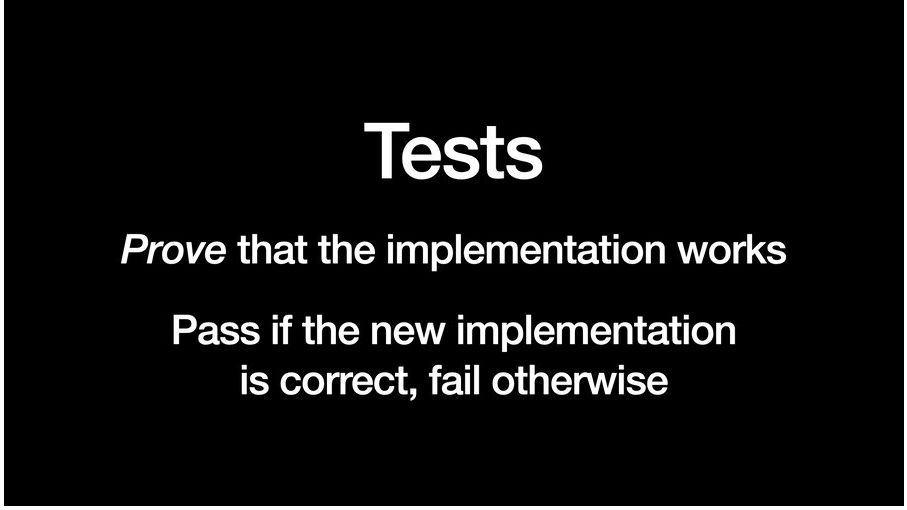
Let's talk about each component in turn.

There's not much to be said about the implementation: your commit should change thing!

It should only change one thing, but what that actually means varies on a case by case basis.

It should be a single change that can be documented, tested and explained independently of other changes.

(Being able to cleanly revert it is a useful property too.)

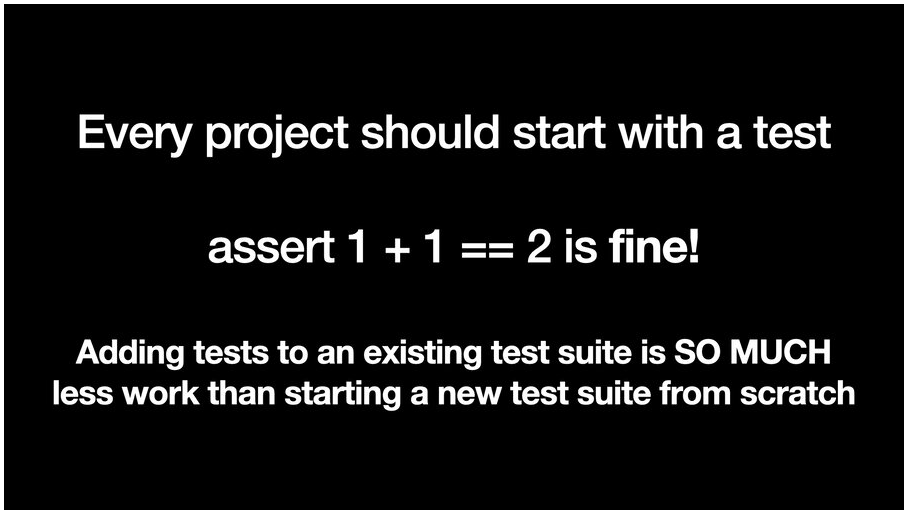


The goals of the tests that accompany a commit are to prove that the new implementation works.

If you apply the implementation the new tests should pass. If you revert it the tests should fail.

I often use `git stash` to try this out.

If you tell people they need to write tests for *every single change* they'll often push back that this is too much of a burden, and will harm their productivity.



But I find that the incremental cost of adding a test to an existing test suite keeps getting lower over time.

The hard bit of testing is getting a testing framework setup in the first place—with a test runner, and fixtures, and objects under test and suchlike.

Once that's in place, adding new tests becomes really easy.

So my personal rule is that every new project starts with a test. It doesn't really matter what that test does—what matters is that you can run `pytest` to run the tests, and

you have an obvious place to start building more of them.

I maintain three cookiecutter templates to help with this, for the three kinds of projects I most frequently create:

- simonw/python-lib for Python libraries
- simonw/click-app for command line tools
- simonw/datasette-plugin for Datasette plugins

Each of these templates creates a project with a `setup.py` file, a README, a test suite and GitHub Actions workflows to run those tests and ship tagged releases to PyPI.

I have a trick for running `cookiecutter` as part of creating a brand new repository on GitHub. I described that in Dynamic content for GitHub repository templates using cookiecutter and GitHub Actions.

This is a hill that I will die on: your documentation must live in the same repository as your code!

You often see projects keep their documentation somewhere else, like in a wiki.

Inevitably it goes out of date. And my experience is that if your documentation is out of date people will lose trust in it, which means they'll stop reading it and stop contributing to it.

The gold standard of documentation has to be that it's reliably up to date with the code.

The only way you can do that is if the documentation and code are in the same repository.

This gives you versioned snapshots of the documentation that exactly match the code at that time.

More importantly, it means you can enforce it through code review. You can say in a PR "this is great, but don't forget to update this paragraph on this page of the documentation to reflect the change you're making".

If you do this you can finally get documentation that people learn to trust over time.

Another trick I like to use is something I call documentation unit tests.

The idea here is to use unit tests to enforce that concepts introspected from your code are at least mentioned in your documentation.

I wrote more about that in Documentation unit tests.

Here's an example. Datasette has a test that scans through each of the Datasette plugin hooks and checks that there is a heading for each one in the documentation.

The test itself is pretty simple: it uses `pytest` parametrization to look through every introspected plugin hook name, and for each one checks that it has a matching heading in the documentation.

The final component of my perfect commit is this: every commit must link to an issue thread.

I'll usually have these open in advance but sometimes I'll open an issue thread just so I can close it with a commit a few seconds later!

Here's the issue for the commit I showed earlier. It has 11 comments, and every single one of those comments is by me.

I have literally thousands of issues on GitHub that look like this: issue threads that are effectively me talking to myself about the changes that I'm making.

It turns out this a fantastic form of additional documentation.

What goes in an issue?

- Background: the reasons for the change. In six months time you'll want to know why you did this.
- State of play before-hand: embed existing code, link to existing docs. I like to start my issues with "I'm going to change this code right here"—that way if I come back the next day I don't have to repeat that little piece of research.
- Links to things! Documentation, inspiration, clues found on StackOverflow. The idea is to capture all of the loose information floating around that topic.

- Code snippets illustrating potential designs and false-starts.
- Decisions. What did you consider? What did you decide? As programmers we make decisions constantly, all day, about everything. That work doesn't have to be invisible. Writing them down also avoids having to re-litigate them several months later when you've forgotten your original reasoning.
- Screenshots—of everything! Animated screenshots even better. I even take screenshots of things like the AWS console to remind me what I did there.
- When you close it: a link to the updated documentation and demo

---

The reason I love issues is that they're a form of documentation that I think of as *temporal documentation*.

Regular documentation comes with a big commitment: you have to keep it up to date in the future.

Issue comments skip that commitment entirely. They're displayed with a timestamp, in the context of the work you were doing at the time.

No-one will be upset or confused if you fail to keep them updated to match future changes.

So it's a commitment free form of documentation, which I for one find incredibly liberating.

I think of this approach as *issue driven development*.

Everything you are doing is issue-first, and from that you drive the rest of the development process.

This is how it relates back to maintaining 185 projects at the same time.

With issue driven development you *don't have to remember anything* about any of these projects at all.

I've had issues where I did a bunch of design work in issue comments, then dropped it, then came back 12 months later and implemented that design—without having to rethink it.

I've had projects where I forgot that the project existed entirely! But I've found it again, and there's been an open issue, and I've been able to pick up work again.

It's a way of working where you treat it like every project is going to be maintained by someone else, and it's the classic cliche here that the somebody else is you in the future.

It horizontally scales you and lets you tackle way more interesting problems.

Programmers always complain when you interrupt them—there's this idea of "flow state" and that interrupting a programmer for a moment costs them half an hour in getting back up to speed.

This fixes that! It's much easier to get back to what you are doing if you have an issue thread that records where you've got to.

Issue driven development is my key productivity hack for taking on much more ambitious projects in much larger quantities.

Another way to think about this is to compare it to laboratory notebooks.

Here's a page from one by Leonardo da Vinci.

Great scientists and great engineers have always kept detailed notes.

We can use GitHub issues as a really quick and easy way to do the same thing!

Another thing I like to use these for is deep research tasks.

Here's an example, from when I was trying to figure out how to run my Python web application in an AWS Lambda function:

Figure out how to deploy Datasette to AWS Lambda using function URLs and Mangum

This took me 65 comments over the course of a few days... but by the end of that thread I'd figured out how to do it!

Here's the follow-up, with another 77 comments, in which I figure out how to serve an AWS Lambda function with a Function URL from a custom subdomain.

I will never have to figure this out ever again! That's a huge win.

---

https://github.com/simonw/public-notes is a public repository where I keep some of these issue threads, transferred from my private notes repos using this trick.

The last thing I want to encourage you to do is this: if you do project, tell people what it is you did!

This counts for both personal and work projects. It's so easy to skip this step.

Once you've shipped a feature or built a project, it's so tempting to skip the step of spending half an hour or more writing about the work you have done.

But you are missing out on *so much* of the value of your work if you don't give other people a chance to understand what you did.

I wrote more about this here: What to blog about.

For projects with releases, release notes are a really good way to do this.

I like using GitHub releases for this—they're quick and easy to write, and I have automation setup for my projects such that creating release notes in GitHub triggers a build and release to PyPI.

I've done over 1,000 releases in this way. Having them automated is crucial, and having automation makes it really easy to ship releases more often.

Please make sure your release notes have dates on them. I need to know when your change went out, because if it's only a week old it's unlikely people will have upgraded to it yet, whereas a change from five years ago is probably safe to depend on.

I wrote more about writing better release notes here.

This is a mental trick which works really well for me. "No project of mine is finished until I've told people about it in some way" is a really useful habit to form.

Twitter threads are (or were) a great low-effort way to write about a project. Build a quick thread with some links and images, and maybe even a video.

Get a little unit about your project out into the world, and then you can stop thinking about it.

(I'm trying to do this on Mastodon now instead.)

Even better: get a blog! Having your own corner of the internet to write about the work that you are doing is a small investment that will pay off many times over.

("Nobody blogs anymore" I said in the talk... Phil Gyford disagrees with that meme so much that he launched a new blog directory to show how wrong it is.)

The enemy of projects, especially personal projects, is *guilt*.

The more projects you have, the more guilty you feel about working on any one of them—because you're not working on the others, and those projects haven't yet achieved their goals.

You have to overcome guilt if you're going to work on 185 projects at once!

This is the most important tip: avoid side projects with user accounts.

If you build something that people can sign into, that's not a side-project, it's an unpaid job. It's a very big responsibility, avoid at all costs!

Almost all of my projects right now are open source things that people can run on their own machines, because that's about as far away from user accounts as I can get.

I still have a responsibility for shipping security updates and things like that, but at least I'm not holding onto other people's data for them.

I feel like if your project is tested and documented, *you have nothing to feel guilty about*.

You have put a thing out into the world, and it has tests to show that it works, and it has documentation that explains what it is.

This means I can step back and say that it's OK for me to work on other things. That thing there is a unit that makes sense to people.

That's what I tell myself anyway! It's OK to have 185 projects provided they all have documentation and they all have tests.

Do that and the guilt just disappears. You can live guilt free!

You can follow me on Mastodon at @simon@simonwillison.net or on GitHub at github.com/simonw. Or subscribe to my blog at simonwillison.net!

From the Q&A:

- You've tweeted about using GitHub Projects. Could you talk about that?
  - GitHub Projects V2 is the perfect TODO list for me, because it lets me bring together issues from different repositories. I use a project called "Everything" on a daily basis (it's my browser default window)—I add issues to it that I plan to work on, including personal TODO list items as well as issues from my various public and private repositories. It's kind of like a cross between Trello and Airtable and I absolutely love it.
- How did you move notes from the private to the public repo?
  - GitHub doesn't let you do this. But there's a trick I use involving a `temp` repo which I switch between public and private to help transfer notes. More in this TIL.

- Question about the perfect commit: do you commit your failing tests?
  - I don't: I try to keep the commits that land on my `main` branch always passing. I'll sometimes write the failing test before the implementation and then commit them together. For larger projects I'll work in a branch and then squash-merge the final result into a perfect commit to main later on.

Posted 26th November 2022 at 3:47 pm · Tagged productivity, talks, testing, documentation · Follow @simonw on Twitter

Source code © 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020 2021 2022