PAGNIs: Probably Are Gonna Need Its

1st July 2021

Luke Page has a great post up with his list of YAGNI exceptions.

YAGNI—You Ain't Gonna Need It—is a rule that says you shouldn't add a feature just because it might be useful in the future—only write code when it solves a direct problem.

When should you over-ride YAGNI? When the cost of adding something later is so dramatically expensive compared with the cost of adding it early on that it's worth taking the risk. On when you know from experience that an initial investment will pay off many times over.

Lukes's exceptions to YAGNI are well chosen: things like logging, API versioning, created_at timestamps and a bias towards "store multiple X for a user" (a many-to-many relationship) if there's any inkling that the system may need to support more than one.

Because I like attempting to coin phrases, I propose we call these **PAGNIs**—short for **Probably Are Gonna Need Its**.

Here are some of mine.

A kill-switch for your mobile apps

If you're building a mobile app that talks to your API, make sure to ship a kill-switch: a mechanism by which you can cause older versions of the application to show a "you must upgrade to continue using this application" screen when the app starts up.

In an ideal world, you'll never use this ability: you'll continue to build new features to the app and make backwards-compatible changes to the API forever, such that ancient app versions keep working and new app versions get to do new things.

But... sometimes that simply isn't possible. You might discover a security hole in the design of the application or API that can only be fixed by breaking backwards-compatibility—or maybe you're still maintaining a v1 API from five years ago to support a mobile application version that's only still installed by 30 users, and you'd like to not have to maintain double the amount of API code.

You can't add a kill-switch retroactively to apps that have already been deployed!

Apparently Firebase offers this to many Android apps, but if you're writing for iOS you need to provide this yourself.

Automated deploys

Nothing kills a side project like coming back to it in six months time and having to figure out how to deploy it again. Thanks to GitHub Actions and hosting providers like Google Cloud Run, Vercel, Heroku and Netlify setting up automated deployments is way easier now than it used to be. I have enough examples now that getting automated deployments working for a new project usually only takes a few minutes, and it pays off instantly.

Continuous Integration (and a test framework)

Similar to automated deployment in that GitHub Actions (and Circle CI and Travis before it) make this much less painful to setup than it used to be.

Introducing a test framework to an existing project can be extremely painful. Introducing it at the very start is easy—and it sets a precedent that code should be tested from day one.

These days I'm all about pytest, and I have various cookiecutter templates (datasette-plugin, click-app, python-lib) that configure it on my new projects (with a passing test) out of the box.

(Honestly, at this point in my career I consider continuous integration a DAGNI—Definitely Are Gonna Need It.)

One particularly worthwhile trick is making sure the tests can spin up their own isolated test databases—another thing which is pretty easy to setup early (Django does this for you) and harder to add later on. I extend that to other external data stores—I once put a significant amount of effort into setting up a mechanism for running tests against Elasticsearch and clearing out the data again afterwards, and it paid off multiple times over.

Even better: **continuous deployment**! When the tests pass, deploy. If you have automated deployment setup already adding this is pretty easy, and doing it from the very start of a project sets a strong cultural expectation that no-one will land code to the `main` branch until it's in a production-ready state and covered by unit tests.

(If continuous deployment to production is too scary for your project, a valuable middle-ground is continuous deployment to a staging environment. Having everyone on your team able to interact with a live demo of your current main branch is a huge group productivity boost.)

API pagination

Never build an API endpoint that isn't paginated. Any time you think "there will never be enough items in this list for it to be worth pagination" one of your users will prove you wrong.

This can be as simple as shipping an API which, even though it only returns a single page, has hard-coded JSON that looks like this:

```
{
  "results": [
    {"id": 1, "name": "One"},
    {"id": 2, "name": "Two"},
    {"id": 3, "name": "Three"}
  ],
  "next_url": null
}
```

But make sure you leave space for the pagination information! You'll regret it if you don't.

Detailed API logs

This is a trick I learned while porting VaccinateCA to Django. If you are building an API, having a mechanism that provides detailed logs—including the POST bodies passed to the API—is

invaluable.

It's an inexpensive way of maintaining a complete record of what happened with your application—invaluable for debugging, but also for tricks like replaying past API traffic against a new implementation under test.

Logs like these may become infeasible at scale, but for a new project they'll probably add up to just a few MBs a day—and they're easy to prune or switch off later on if you need to.

VIAL uses a Django view decorator to log these directly to a PostgreSQL table. We've been running this for a few months and it's now our largest table, but it's still only around 2GB—easily worth it for the productivity boost it gives us.

(Don't log any sensitive data that you wouldn't want your development team having access to while debugging a problem. This may require clever redaction, or you can avoid logging specific endpoints entirely. Also: don't log authentication tokens that could be used to imitate users: decode them and log the user identifier instead.)

A bookmarkable interface for executing read-only SQL queries against your database

This one is very much exposing my biases (I just released Django SQL Dashboard 1.0 which provides exactly this for Django+PosgreSQL projects) but having used this for the past few months I can't see myself going back. Using bookmarked SQL queries to inform the implementation of new features is an incredible productivity boost. Here's an issue I worked on recently with 18 comments linking to illustrative SQL queries.

(On further thought: this isn't actually a great example of a PAGNI because it's not particularly hard to add this to a project at a later date.)

Driving down the cost

One trick with all of these things is that while they may seem quite expensive to implement, they get dramatically cheaper as you gain experience and gather more tools for helping put them into practice.

Any of the ideas I've shown here could take an engineering team weeks (if not months) to add to an existing project—but with the right tooling they can represent just an hour (or less) work at the start of a project. And they'll pay themselves off many, many times over in the future.

Posted 1st July 2021 at 7:13 pm · Follow me on Mastodon or Twitter or subscribe to my newsletter

More recent articles

- Weeknotes: datasette-enrichments, datasette-comments, sqlite-chronicle - 8th December 2023
- Datasette Enrichments: a new plugin framework for augmenting your data - 1st December 2023
- llamafile is the new best way to run a LLM on your own computer - 29th November 2023
- Prompt injection explained, November 2023 edition - 27th November 2023
- I'm on the Newsroom Robots podcast, with thoughts on the OpenAI board - 25th November 2023
- Weeknotes: DevDay, GitHub Universe, OpenAI chaos - 22nd November 2023
- Deciphering clues in a news article to understand how it was reported - 22nd November 2023
- Exploring GPTs: ChatGPT in a trench coat? - 15th November 2023

- Financial sustainability for open source projects at GitHub Universe - 10th November 2023
- ospeak: a CLI tool for speaking text in the terminal via OpenAI - 7th November 2023

djangosqldashboard 10 githubactions 39 continuousdeployment 10 continuousintegration 16 yagni 5 pagni 3 pytest 13 softwareengineering 47

Next: Django SQL Dashboard 1.0

Previous: Weeknotes: sqlite-utils updates, Datasette and asgi-csrf, open-sourcing VIAL