Deployment

UNC COMP 523

Mon Sep 21, 2020

Prof. Jeff Terrell

Announcements

- music: *&Run* by Sir Sly - just for fun :-)
- I've repurposed TA Yiyuan (Bill) Li, who will not be holding office hours any more
- for technical help, remember the App Lab
- also remember Piazza and my office hours (Tuesdays from 10am-12pm)

On software

> Software is eating the world.
> – Marc Andreessen, WSJ editorial, Aug. 2011 (source)

- software is disrupting industry after industry
- those who leverage software effectively are outcompeting those who do not
- (isn't it nice to be the music makers?)
- further, competitive effectiveness is often determined by how *well* a company leverages software
- "well" here means (approximately) "in an agile way"
  - write it faster
  - fix it faster
  - pivot faster, as needed

Think about it:

What *is* software?

On the essence of software

- software is a set of instructions
- it's some bits stored on a hard disk somewhere
- *it's inert*
- software needs to be *animated* to be useful
- we'll call this animating force a *system*

Agility of the system

- the most effective software creators and users understand not just agility of software but also *agility of system*
- this is the core idea of class today: how to unlock *system-level* agility

Examples of system agility

- all the big-name tech companies exemplify this agility
- (is it correlation or causation?)
- key metric for this agility: number of production deploys per day

Amazon announced it deployed

130,000 times per day

...

Amazon announced it deployed

130,000 times per day

...in 2015!

Examples of system agility (cont.)

- all the big-name tech companies exemplify this agility
- (is it correlation or causation?)
- key metric for this agility: number of production deploys per day
- Amazon announced it deployed 130,000 times per day...in 2015!
- Google has also exceeded 10k deploys/day for a while
- employees tend to be happier, too
- (consider the stress of having one huge deploy every quarter)
- this idea of system-level agility is the key idea behind "DevOps"
- more (fun) reading: *The Phoenix Project*

Outline

- ~~intro: why system agility matters~~
- how: the twelve-factor app
- demo: using Heroku

Outline

- ~~intro: why system agility matters~~
- **how: the twelve-factor app**
- demo: using Heroku

How to enable system agility?

Driving question: how do we enable system agility?

Answer: follow the recommendations of the twelve-factor app (https://12factor.net/)

A note about the target audience

- how do desktop apps get deployed? by sticking an installer package on a web site.
- how do mobile apps get deployed? by uploading a compiled package to the app store.
- neither of these approaches provides much opportunity for system agility; we will ignore them.
- we're mainly concerned here with deploying backend code
- if you run a web app, your frontend can go along for the ride, as it's served by the same backend
- if you're using a backend service like Firebase, you don't need to deploy a backend
- (hopefully those of you without backend code will still find this interesting)

The twelve-factor app overview

- set of 12 principles for deploying your app
- implications for how the app runs in production
- implications for how you develop your app locally
- key idea: have a *clean contract* with the operating system to enable easy portability of your code to multiple computers
- let's walk through the 12 principles

1. One codebase (tracked in e.g. git), many deploys

- a *codebase* is like a database of code, e.g. a git repository
- each deploy is a running *instance* of the app that comes from *the same codebase*
- the obvious deploy: a production ("prod") site, i.e. where user will access the service
- an optional deploy: a staging site, where developers or testers can preview features before they hit production
- other needed deploys: a local development ("dev") system, one per developer
- note: each dev instance should be isolated from all others, for maximum confidence and agility
- again, prod and dev (and any others) come from the *same codebase*
- antipattern: separate repos for frontend and backend

2. Explicitly declare and isolate dependencies

- problem: same code runs on one system but not on another
- typical cause: implicit dependencies installed on one but not the other
- solution:
  - explicitly list all dependencies of your app (e.g. in a `package.json` or `Pipfile` file)
  - isolate your app, so that *only* listed dependencies are accessible
- most common type of dependency: something listed in your manifest, e.g. an NPM or Python package
- be careful about system dependencies, e.g. `curl`, `pandoc`, or something installed by ImageMagick
  - one approach: *vendor* the tools into the repo, so that the tools are installed locally at build time
  - another approach: use docker to be explicit about these deps too

3. Store config in the environment

> But if we run the same code for every deploy, how can anything be different?

- answer: by separating the *config* from the codebase
- config is accessible via *environment variables*, provided by the operating system to your process
- environment variables have a name and a string value
- example: `PORT` or `DATABASE_URL`
- what goes in the config? *anything that's able to vary between deploys*
- so if your framework makes you store a list of routes in a config file, that's a different kind of config
- how do we keep it out of the codebase? by ignoring it, e.g. `echo .env > .gitignore`
- key idea: **config varies across deploys; code varies across features**

4. Treat backing services as attached resources

- essentially, access services like databases by URL
- store needed URLs in config, not code
- if needed, services can be detached and attached at will
  - if a particular database service is acting up, spin up a new one and update the config
  - if you need to connect to the prod database service to fix a data issue, you can do so with your local app
- antipattern: hardcoded URLs in your code
- antipattern: embedded databases like SQLite or H2
- see diagram

5. Strictly separate build and run phases

- 3 stages of transforming a codebase into a (non-development) deploy:
  1. *build*: create an executable bundle from the code
  2. *release*: combine the build with a config
  3. *run*: animate the release
- see diagram
- if code changes, re-build and re-release
- if config changes, just re-release
- releases are *versioned* (e.g. `v1`, `v123`) and *immutable*
- can *roll back* to an earlier version as needed
- build happens in foreground, synchronously, so *make errors happen there* if possible (cf. static languages)
- run happens in background, maybe in the dead of night
- don't be a "cowboy", patching prod systems directly

6. Execute the app as one or more stateless processes

- systems can and do reboot often
- then anything stored in memory gets wiped
- antipattern: inter-process communication, which couples components and hinders scalability
- antipattern: sticky sessions, where a single client gets routed to the same server every time

7. Export services via port binding

- bind to the port given by the `PORT` environment variable
- don't expose your services in any other way

8. Scale out via the process model

- if you need to scale, do it by scaling to more stateless processes

9. Maximize robustness with fast startup and graceful shutdown

- systems can get rebooted every day; make startup and shutdown fast and error-free
- builds can be slow, but the run stage should be fast

10. Keep development and production as similar as possible

- problem: app works locally, but fails in production
- cause: some difference between the production environment and your local one (see #2)
- you don't want to be debugging these kinds of issues!
- solution: keep "dev" and "prod" as similar as possible, a.k.a. "dev/prod parity"
- philosophy: reduce the gaps between dev and prod:
  - the time gap, where deploys to production happen much later than the development of that commit
  - the personnel gap, where devs write code and "ops" people deploy it
  - the tools gap, where different tools exist in dev and prod
- this is the main reason why we keep code and config separate

11. Treat logs as event streams

- some logging systems can complicate systems issues
- suggestion: simply print notable events to standard output and let the system timestamp and aggregate your messages
- makes observation during development easy
- but is also valuable during production

12. Run admin/management tasks as one-off processes

- use the same codebase and same config as the service you typically deploy
- use a different entry point to the code to run management tasks
- examples:
  - migrate the database schema to a new version
  - launch a console to inspect the prod database and run live code

Twelve-factor app summary

- *key goal: agility of systems*
- key takeaways for app builders:
  - get config (incl. `PORT` and `DATABASE_URL`) from environment variables
  - keep config out of the codebase ("ignore" it in git)
  - use the same tools locally and in production

Outline

- ~~intro: why system agility matters~~
- ~~how: the twelve-factor app~~
- demo: using Heroku

Heroku

- Heroku embodies the 12 factors
- (the 12 factors were written by Heroku people)
- it's not the only place to deploy your code, but it's an excellent choice if you don't already have opinions or experience
- their tagline "optimized for developer experience" rings true to my experience
- plus: nice free tier (incl. many addons with a free tier)
- note: they will hibernate your free tier "dyno" (server) after ~30 minutes of inactivity
- also restart your dyno every 24 hours
- have many "buildpacks" to create builds for common approaches
- also lots of addons (incl. databases, email sending, etc)
- demo: a GitHub-connected pipeline complete with review apps (from this repo)

Take-away questions