Lecture Notes: *Everything as Code*

Origins of *Everything as Code*

It's become a core DevOps tenet: *wherever possible, treat it as code*.

This idea arose from *Infrastructure as Code* which is mentioned in *The DevOps Handbook*, chapter 1:

> DevOps also extends and builds upon the practices of infrastructure as code, ...
>
> **the work of Operations is automated and treated like application code, so that modern development practices can be applied to the entire development stream.**
>
> This further enabled fast deployment flow, including continuous integration, continuous delivery, and continuous deployment.

The Evolution of Infrastructure as Code

Discuss the evolution of abstraction from physical machines to virtual machines which allowed infrastructure to be expressed as [mostly] human-readable code. This in turn permitted the automated configuration and maintenance of IT infrastructure.

The Place of Version Control

Version control, *shared* version control is key.

Humble *et al.*, *Why Enterprises Must Adopt DevOps to Enable Continuous Delivery*, Cutter IT Journal **24**, 6 (2011).

Version Control

- It's fundamental.
- git is not a great tool, but we're stuck with it.
- GitLab provides many features to ease the use of git and to provide safeguards.

> Git allows a wide variety of branching strategies and workflows. Because of this, many organisations end up with workflows that are too complicated, not clearly defined, or not integrated with issue tracking systems. Therefore, we propose GitLab flow as a clearly defined set of best practices. It combines feature-driven development and feature branches with issue tracking.

GitLab Flow https://docs.gitlab.com/ee/topics/gitlab_flow.html as a solution.

DevOps aphorism: **Take care of your repository and it will take care of you.**

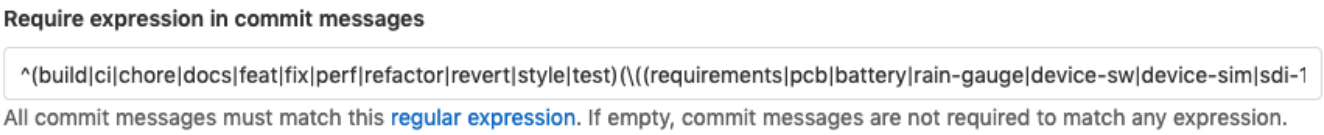
Commit Message Standards

Examples:

- https://github.com/angular/angular/blob/master/CONTRIBUTING.md#commit
- https://manual.limesurvey.org/Standard_for_Git_commit_messages

Use:

1. GitLab push rules `Settings` → `Repository` → `Push rules`

2. `gitlint`

Branching Strategy

Must have one! *The DevOps Handbook*, Ch. 11, describes a fundamental tension regarding branching:

Optimise for individual productivity vs. **Optimise for team productivity**

The problems with the former are merge "hell" and integration "hell", where significant effort and rework is required to integrate work on long-lived individual branches. The problems with the latter are the risk of breaking the `main` branch from which the software running in production is derived.

Git Flow attempted to address the problems, but became very complicated (due in part to the lack of structure in the design of `git`; as a tool it will allow *almost anything* to occur). GitLab Flow https://docs.gitlab.com/ee/topics/gitlab_flow.html seeks to address the problem with an emphasis on short-lived feature branches. [Not mentioned in the lecture but still important: GitLab's protected branch defaults match the GitLab Flow paradigm, where committing to `main` is deprecated.]

There are other problems which can be caused, often inadvertently. For example, using `git rebase` to rewrite the commit history on a branch which has not been shared with others (i.e. "pushed") has many benefits... but rewriting the commit history on a branch which others have can cause them a *lot* of trouble. See: Recovering from Upstream Rebase in the [ridiculously long] `git rebase` manual page https://git-scm.com/docs/git-rebase:

> Rebasing (or any other form of rewriting) a branch that others have based work on is a bad idea: anyone downstream of it is forced to manually fix their history. This section explains how to do the fix from the downstream's point of view. **The real fix, however, would be to avoid rebasing the upstream in the first place.**

[Note: the following was discussed in passing in the lecture] Branching and merging practices which are standard practice in other version control systems, such as Subversion, can cause problems in `git`. For example, using `git merge main` into a feature branch in `git` this will create a "Foxtrot" merge, see: *Protect our Git Repos, Stop Foxtrots Now!* https://blog.developer.atlassian.com/stop-foxtrots-now/. The key takeaway from this should be:

> ... always do `git pull –rebase` and ... never type `git merge main` ...

The former will also avoid "merge commits" and spurious conflicts; in `git` you should never be merging `main` into another branch.

Linting

Linting is a form of static analysis. It checks not only language syntax but also deviations from programming and style standards.

- Shared standards — automatic enforcement for the benefit of consistency.

pre-commit

pre-commit https://pre-commit.com/ is a framework for automatically maintaining `git` pre-commit hooks *and* the packages which are called from them:

> We built pre-commit to solve our hook issues. It is a multi-language package manager for pre-commit hooks. You specify a list of hooks you want and pre-commit manages the installation and execution of any hook written in any language before every commit.

Note added: pre-commit even has a `protect-first-parent` hook which protects the repository from Foxtrot merges (see: https://pre-commit.com/hooks.html).

Conclusions

The secret is that treating everything as code is about the DevOps culture of minimising manual toil and collaboration through knowledge sharing.

Epilogue

GitOps https://about.gitlab.com/topics/gitops/
