# Git Strategy

TLDR;

We have setup a single git repo to host all the microservices and shared libraries code. We chose trunk-based dev as we try to have short-lived feature branches.

# Process Approach

Essentially there are two approaches:

* Git flow
* Github flow (aka trunk-based development)

The general idea behind git-flow is to have several separate branches that always exist, each for a different purpose: e.g., master, develop, feature, release, and hotfix. The process of feature or bug development flows from one branch into another before it’s finally released.

Git Flow may be worth it when:

* When you have discrete named or numbered releases
* When you need to freeze development on a release candidate while still continuing to develop and integrate features for a subsequent release
* When multiple versions of the software need to be supported and maintained independently

With Git flow it is definitely harder to deal with in a continuous deployment model.

GitHub Flow is a simplified alternative to Git flow. GitHub has some of the same elements as Git Flow, such as feature branches. But unlike Git Flow, GitHub Flow combines the mainline and release branches into a “main” and treats hotfixes just like feature branches. This simplified model is better suited to continuous delivery models where changes can be quickly made and easily deployed, sometimes multiple times a day. Microsoft use a trunk-based development flow called Release Flow.

References

[Trunk-Based Development](https://trunkbaseddevelopment.com/)

[Microsoft Engineering and Release Flow](https://docs.microsoft.com/en-us/azure/devops/learn/devops-at-microsoft/release-flow)

# Git Repo Approach

When you start moving to microservices, the first question you ask is: How do I organise the codebase? Do you create a repository for each service, or do you create a single ‘mono repo’ for all services? The two approaches are:

* “Monorepo” – each microservice has its own folder in a single repository
* “Multirepo” - each microservice has its own dedicated git repository

Obviously, there are pros and cons to each approach. If you need to restrict access to only certain repos then multirepo is the way to go - but we don't have this security requirement. Setting up DevOps CI and CD is with multirepo is much easier was you let each repository have its own process for being deployed. When using a monorepo, there needs to be additional logic for sorting through the folders that make up the different projects within the monorepo. This is really setting up a trigger on a folder rather than the repo file set as a whole. For multirepos: There is clear ownership and smaller codebases are easier to manage. If a codebase gets too big then git clones, pulls, and pushes can take too much time. The downside is that the code base does not match the architecture and you want every engineer to understand the bigger picture. Additionally, it could be more challenging for us to enforce standardisation of code across repositories. We really want one team with knowledge across all of the services. For me, the decision comes down to not wanting to git clone a whole bunch of different repositories at one time (20?). Apparently, Google uses a monorepo approach… A monorepo approach feels like the right solution for us.

\*Not quite true. You can have sub-repos, but imho, for us, it is an unnecessarily complication. - ‘KISS’

References

[Journey to Microservices](http://blog.shippable.com/our-journey-to-microservices-and-a-mono-repository)

[Repo Style Wars – Mono v Multi](http://www.gigamonkeys.com/mono-vs-multi/)

[Monorepos in Git](https://www.atlassian.com/git/tutorials/monorepos) (Atlassian)