A Guide to Git

- **OBJECTIVE:** *Improve workflow with nvim, fugitive, lazygit*
- **References:**
  - Fugitive workflow @Youtube-preview
  - Config your git @Youtube

Git Operations

Understand the fundamentals of the git tool

Git config

There are two locations a git config file can exist:

- **Local** git repo: in the repo itself (and ignored by `.gitignore`)
- **Global** exist outside of the repo e.g. gitconfig

Git merge vs re-base vs squashing

There are different strategies to merge two `branches`:

- **Merging -** Merge and ties together the `history` of both `branches`
- **Re-basing -** Merge the `commits` of a branch to the **tip** of `main` and then performs a `fast-forward-merge`
  - **Fast-forward -** `merge` and move the pointer of the target branch forward to the latest commit of the course branch *avoiding the need to create a merge commit*
- **Squashing -** Squeeze all the `commits` of `feature branch` into one single `commit` and merge it with `main`

Git commit vs stash

Visit:* Git Stash vs Git Commit

Write better commit messages

- **Goal(s):** Write better and more comprehensive commit messages
- **Stumble upon:** https://www.freecodecamp.org/news/how-to-write-better-git-commit-messages/

The Anatomy of a Commit Message

- **Basic:** `git commit -m <message>`
- **Detailed:** `git commit -m <title> -m <description>`

5 Steps to write a better commit message

- Note taken on [2024-09-21 Sat 21:37]
  To come up with thoughtful commits consider the following:
  - Why have I made these changes?
  - What effect have my changes made?
  - Why was the change needed?
  - What are the changes in reference?

1. **Capitalization and Punctuation:** Capitalize the first word and do not end in punctuation. If using *Convention Commits*, remember to use all lowercase
2. **Mood:** Use imperative mood in the subject line. **Example -** `Add fix` for `dark mode toggle state`. Imperative mood fives the tone you are giving in *order* or *request*
3. **Type of Commit:** Specify the type of commit. It is recommended and can be even more beneficial to have consistent set of words to describe your changes. **Example:** *Bugfix*, *Update*, *Retractor*, *Bump*, and so on.
4. **Length:** The first line should ideally be no longer than 50 chars, and the body should be restricted to 72 chars
5. **Content:** Be direct, try to eliminate fillers words and phrases in theses sentences (**Example:** though, maybe, I think, kind of), Think like a journalist

Conventional Commits

Conventional Commit is a formatting convention that provides as set of rules to formulate a consistent commit message structure like so:

```
<type>[option scope]: <description>

[optional body]

[optional footer(s)]
```

The commit `type` can include the following:

- `feat` -a new feature is introduces with the changes
- `fix` -a bug fix has occurred
- `chore` -changes that do not relate to a fix or feature and don't modify src or test files (for example updating dependencies)
- `refractor` -refactored code that neither fixes a bug nor adds a feature
- `doc` - updates to documentation such as the README or other markdown files
- `style` -changes that do not affect the meaning of the code, likely realted to code formatting such as white-space, missing semi-colons, and so on
- `test` -including new or correcting previous tests
- `perf` -performance improvements
- `ci` -continuous integration related
- `build` -changes tha affect the build system or external dependencies

- `revert` -reverts a previous commit

The commit type subject line should be all lowercase with a character limit to encourage succinct descriptions

The `[optional commit body]` should be used to provide detail that cannot fit within the character limitations of the subject line description

It is also good location to utilize `BREAKING CHANGE: <description>` to note the reason for a breaking change within the commit

The `[footer]` is also optional. We use the footer to link the JIRA story what would be closed with these changes for example: `Closes D2!I-<JIRA #>`

```
fix: fix foo to enable bar

This fixes the broken behavior of the component by doing xyz.

BREAKING CHANGE
Before this fix foo wasn't enabled at all, behavior changes from <old> to <new>

Closes D2IQ-12345
```

- **NOTE** The ensure that these committing conventions remain consistent across developers, commit message linting can be configured before changes are able to be pushed up. Commitizen is a great tool to enforce standards, sync up semantic versioning, along with other helpful features

Clean commit history

- **OBJECTIVE:** Clean up git commit history
- **Reference:** Git tools rewrite history @Doc-git

Securely storing secrets in git

- **OBJECTIVE:** Learn how to securely store secrets in git

Managing A Nested Git Project

For an open-source organization, it can be tricky to achieve single-source `documentation` and `dependency management` for the community and the product. The `documentation` and `project` often end up fragmented and redundant, which makes them difficult to maintain

https://opensource.com/article/20/5/git-submodules-subtrees

- **Reference:** How to create a nested project @Overflow

Git Submodules Approach

- Git `submodules` are *git repositories* within a *git repository*
  - The `submodule` are pointers which points to a **specific commit** of the *child repository*
  - `Submodule` can be *nested* - meaning you can have a `submodule` of a `submodule`
- This approach meaning you can have a working tree of submodules as a working directory tree

  [!WARNING] `submodules` are pointer to s specific commit meaning having too many layers of them can obstruct workflow as you will have to update each `submodule` along with its new pointers

1. Clone and Load Submodules

   Downloading `submodules` sequentially can be a tedious task, so `clone` and `submodule update` will support the `--jobs` or `-j` parameter

   ```
   git submodule update --init --recursive -j8
   git clone --recursive --jobs 8 <URL to git repo>
   ```

2. Add Submodules

   - To add a child repository to parent repository:

     ```
     git submodule add <URL to Git repo>
     ```

     - To create an empty repository on remote (using `github-cli`)

       ```
       gh repo create <Repo name> --public
       ```

   - To Initialize an existing Git submodule

     ```
     git submodule init
     ```

3. Remove a Submodule

   Merely deleting a child project manually won't remove the child project from the parent repository as it is staged. To delete a *child repository* run:

   ```
   git rm -rf submodule
   ```

4. Pull submodules

   Before building or running the *parent repository*, you have to make sure that the child *dependencies* are up to date

   - To pull all changes in submodules:

     ```
     git submodule update --remote
     ```

5. Make Changes to Submodules

As mentioned above `submodules` are pointer to a specific `commit` of a *repository*. Thus in order to make changes in a `submodule` (from parent repository) you have to first checkout to an existing `branch` as opposed to a `commit`

> [!TIP] `git` support running command from outside of the *working directory* using command `git -C path/to/repo <command>`

- To checkout a submodule:

  ```
  git -C path/to/submodule checkout main # or any preferred branch
  ```

- Then edit on the `submodule` like a *standalone repository*
  - To commit changes

    ```
    git -C path/to/submodule commit -m "<message>" <files>
    ```

  - To push changes of a `child-repo` from the `parent-repo`

    ```
    git -C /path/to/submodule push origin main # or any preferred branch
    ```

  - To pull changes to `child-repo`, using the same command

    ```
    git -C /path/to/submodule/ pull
    ```

  > [!NOTE] Git support running command for each initialized *child repository* using `git submodule foreach <command>`. However this run `commands` in sequence thus takes up a lot of time to finish all commands on every `submodules`. In stead running `commands` in parallel might take more overhead but less time

Symlinks Approach

Managing a *nested project* can be a tedious tasks. Especially when the `child-repo` contains different `child-repos`. This create a layer of index to different pointer to maintain when making changes to any `child-repo`. Thus instead of create a nested level, use `symlink` to create a list of local pointer the stay fixed even when a `child-repo` is changed

```
ln -s absolute/path/to/submodule -t target/dir
```