JJ Reference

Model

A Jujutsu repository is a DAG (directed acyclic graph) whose nodes are called ***changes***. Each change has:

- A state of the filesystem within the repository directory. You can imagine each change storing a full copy of the directory and all the files in it, though `jj` is more efficient than this.
- File ***conflicts***. Some files in a change may contain conflicts, from a variety of different sources. These conflicts are local to the change. (Unlike `git`, they do not block your use of `jj`.)
- One or more ***parent*** changes. (Though there is a root change which has no parents and always has an empty directory).
- A textual ***description*** of the change, a.k.a. a commit message. This is always present, but defaults to the empty string.

There is some additional information attached to the DAG:

- Exactly one of the changes is the ***working change***, a.k.a. `@`. The docs call this the "working copy revision". (This is analogous to `git`'s `HEAD`.)
- There may be some ***bookmarks***, which are unique string labels on changes. (When interfacing with `git`, these bookmarks act as branch names.)
- The repository may also be linked to a ***remote repository*** (e.g. Github). If so, when `push`ing and `fetch`ing, `jj` records the ***last known position*** of each remote bookmark, written `BOOKMARK@REMOTE` (e.g. `feat-ui@origin`).

Most `jj` commands modify your local repository DAG in some way. Some general rules will help you predict how it responds to modifications:

- When you make `@` point at a change, your repository directory is updated to match that change's files.
- If you delete the change that `@` is pointing at, `@` moves to a new empty change off of its parent(s).
- If a change has no file modifications and no description, and is not referenced by `@` or by a bookmark, it disappears silently into the night.
- A change represents a diff. Moving a change tries to apply the diff to its new parent, but this may cause merge conflicts.
- Many commands act on `@` by default. Almost all of them can take a `-r`/`--revision` argument to act on a different change.

File Conflicts

If the working change (`@`) has a ***file conflict***, resolving it is as simple as editing the file so as to no longer have conflict markers (`<<<<<<<`, `>>>>>>>`, etc.) in it. For a binary file, replace the file with the version you want. `jj restore` may be useful for this purpose. (Unlike `git`, file conflicts don't block you.)

jj git push

`jj git push` copies changes from the local repo into the remote repo. If a local change has been modified since it was last pushed, it becomes a brand new change in the remote repo (just like force pushing in `git` replaces old commits with new commits). To prevent you from accidentally doing this to `main`, `jj git push` makes all pushed changes to the primary branch immutable. You can still edit them if you want, but you have to pass the `--ignore-immutable` flag.

All local bookmarks are similarly copied to the remote repo. However, if a bookmark is present both locally and remotely, `jj` checks if its (locally recorded) ***last seen position*** matches its current position in the remote repo. If so, the bookmark's position in the remote repo is updated. If not, this command fails and tells you to `jj git fetch` first (because it means that someone else updated the bookmark since you last pushed it).

jj git fetch

`jj git fetch` copies changes from the remote repo into the local repo. If a change has been modified in the remote repo, it turns into a new change locally. Though most of the time you're just fetching fresh new changes.

Local bookmarks are advanced to match the change that they're on in the remote repo. However, if the change a bookmark is at in the remote is not a descendant of the change it's on locally, `jj git fetch` creates a second copy of that bookmark. This is called a ***bookmark conflict*** because it violates the invariant that bookmark names are unique. (This is analogous to the situation where `git pull` produces a merge conflict.) It is up to you how to resolve this "bookmark conflict":

- If you want to merge the two changes, say `jj new CHANGE-ID-1 CHANGE-ID-2`, resolve any file conflicts, then update the bookmark with `jj bookmark set BOOKMARK-NAME`. (You can get the change ids by running `jj bookmark list BOOKMARK-NAME`.)
- If you want to discard one of the two changes and just use the other one, say `jj bookmark set CHANGE-ID` for the change you want to keep.
- If you want to rebase one of the changes to come ***after*** the other, say `jj rebase -b SECOND-CHANGE-ID -d FIRST-CHANGE-ID`, then `jj bookmark set BOOKMARK-NAME`. This will rebase not only the second change itself, but all changes after it forked away from the first change.

Commands

Global Setup Commands

- `jj config set --user name MY_EMAIL`. Set your name for signing commits.
- `jj config set --user email MY_EMAIL`. Set your email address for signing commits.
- `jj config edit --user`. Manually edit the configuration file.

Instead of `--user`, you can pass `--repo` to change the repository specific config, which takes priority.

Repository Commands

- `jj git init`, or `jj git clone URL [DESTINATION]`. Make or clone a git-backed repo.
- `jj git init --colocate`. Make an existing `git` repo also be a `jj` repo.

Editing your Local Repo

The attached JJ Cheat Sheet visually describes the most common/fundamental commands for editing a `jj` repo.

There are also a couple of "alias" commands that are best thought of as combinations of other `jj` commands:

- `jj commit`. Shorthand for `jj describe; jj new`.
- `jj bookmark set BOOKMARK`. Either `create` or `move` the bookmark, whichever is valid.