Introduction

Hi there, I'm Steve. This is a tutorial for Jujutsu—a version control system.

This tutorial exists because of a particular quirk of mine: I love to write tutorials about things as I learn them. This is the backstory of TRPL, of which an ancient draft was "Rust for Rubyists." You only get to look at a problem as a beginner once, and so I think writing this stuff down is interesting. It also helps me clarify what I'm learning to myself.

Anyway, I have been interested in version control for a long time. I feel like I am the only person alive who doesn't mind `git` 's CLI. That's weird. I also heard friends rave about "stacked diffs" but struggled to understand what exactly was going on there. Nothing I read or conversations I had have clicked.

One time I was talking with Rain about this and she mentioned `jj` being very cool. I looked at it but I didn't get it. I decided to maybe just come back to this topic later. A very common thing people have when learning Rust is, they'll try it, quit in frustration, and then come back months later and it's easy and they don't know what they were struggling with. Of course, some people never try again, and some who try don't get over the hump, but this used to happen quite often, enough to be remarkable. So that's what I decided with `jj` . I'd figure this stuff out later.

Well, recently Chris Krycho wrote an article about jj. I liked it. I didn't fully grok everything, but it at least felt like I could understand this thing finally maybe. I didn't install it, but just sat with it for a while.

I don't know if that is why, but I awoke unusually early one morning. I couldn't get back to sleep. Okay. Fine. Let's do this. I opened up the official jj tutorial, installed `jj` , made this repository on GitHub, followed the "cloning a git repo" instructions to pull it down, and started this mdBook.

What follows is exactly what the title says: it's to teach me `jj` . I am publishing this just in case someone might find it interesting. I am very excited about `jj` . I feel like I'm getting it. Maybe your brain will work like mine, and this will be useful to you. But also, it may not. I make no claim that this tutorial is good, complete, or even accurate.

You can find the source for this content here. Feel free to open issues or even send me pull requests for typos and such. Zero guarantees I will respond in a timely fashion though.

Anyway, let's get on with it.

What is jj and why should I care?

`jj` is the name of the CLI for Jujutsu. Jujutsu is a DVCS, or "distributed version control system." You may be familiar with other DVCSes, such as `git`, and this tutorial assumes you're coming to `jj` from `git`.

So why should you care about `jj`? Well, it has a property that's pretty rare in the world of programming: it is both *simpler* and *easier* than `git`, but at the same time, it is *more powerful*. This is a pretty huge claim! We're often taught, correctly, that there exist tradeoffs when we make choices. And "powerful but complex" is a very common tradeoff. That power has been worth it, and so people flocked to `git` over its predecessors.

What `jj` manages to do is create a DVCS that takes the best of `git`, the best of Mercurial (`hg`), and synthesize that into something new, yet strangely familiar. In doing so, it's managed to have a smaller number of essential tools, but also make them more powerful, because they work together in a cleaner way. Furthermore, more advanced `jj` usage can give you additional powerful tools in your VCS sandbox that are very difficult with `git`.

I know that sounds like a huge claim, but I believe that the rest of this tutorial will show you why.

There's one other reason you should be interested in giving `jj` a try: it has a `git` compatible backend, and so you can use `jj` on your own, without requiring anyone else you're working with to convert too. This means that there's no real downside to giving it a shot; if it's not for you, you're not giving up all of the history you wrote with it, and can go right back to `git` with no issues.

How to read this tutorial

This tutorial is split up into a few different sections. While I make sure that each concept is introduced before its used, if you're okay with looking some stuff up on your own, you can skip around if you'd like.

This tutorial is *intended* to be read from front to back. You can even follow along with the examples if you'd like. But I'm not your dad, you can do whatever you'd like.

Here's what each section is about:

Hello, world!

In this section, we show off the very core concepts of `jj` : repositories, changes, looking at history, stuff like that. This stuff is simple but also very important! Everything else is built on top of this.

If you want to skip around, I would at least consider reading this part first. Okay, just skim. Point is, if you don't already know this stuff, you'll have a harder time understanding later sections.

Real-World Workflows

The workflow we developed in "Hello, world!" works, but isn't as pleasant as it should be. There are two basic workflows that are used by the majority of `jj` enthusiasts, and so we go over both of them in detail.

Branching, merging, and conflicts

Once you've got a core workflow down, the next concept involves branching, and how to merge branches. Once you do that enough, you'll also have to learn how to resolve merge conflicts.

`jj` natively is a "branchless" VCS, which sounds especially weird coming from `git`. If you've ever wondered what that's like, this part will show you how, and if you're wondering how you interact with `git`'s named branches, well, you're getting ahead of me! That's next!

Sharing your code with others

Because you'll be sharing your code with `git` users, you'll need to understand how to use named branches in `jj`. We'll cover that, and then also share how you can upload your code to `GitHub` (or any other remote repository, really) and even respond to pull request feedback that upstream may give you.

More advanced workflows

Now that we have a very solid understanding of `jj`, we can start to do very cool things. Up until now, we've been fitting `jj` into `git` shaped workflows, to make it easier to understand. This section will show you the power of `jj`'s various primitive operations by some practical workflows you may decide to incorporate into your usage of `jj`.

Additionally, we'll cover some advanced features of the tool: workspaces and colocated repositories. You may not use these, but it's good to know about them.

Fixing problems

At some point, you'll probably make some sort of mistake. Maybe you combine two commits you didn't intend to combine. Maybe you tried to fix a merge conflict, but you don't like where you ended up.

If you've ever used `git reflog` , you know exactly what this chapter is about. But if you don't, well, you know how some people say "you can't ever lose your work in `git` " even though you absolutely can? This section will show you why it's even more difficult to do so in `jj` .

Customizing your experience

Finally, `jj` is very customizable. We'll talk about how you can customize various parts of `jj` , including its very powerful templating system for showing exactly the output you want from any command.

Hello, world!

Let's create a new repository with `jj`, and show off some basic commands. Here's what we're going to learn:

- How to install `jj`
- Creating a repository with `jj git init`
- Viewing the current status with `jj st`
- Using `jj describe` to describe our commits in a human-friendly way
- Using `jj new` to create new changes
- Viewing the contents of your repository with `jj log`

How to install `jj`

This tutorial is written when `jj` is at version 0.23.0. It may work for later versions, but you also may need to adapt.

For the full range of ways to install `jj`, you can visit the Installation and Setup page of the official documentation. Personally, because I am a Rust developer, and `jj` is written in Rust, I installed my copy like this:

```
$ cargo install jj-cli@0.23.0 --locked
```

If you're not a Rust developer, please read the documentation to figure out how to install things on your platform; I could replicate that information here, but I'm not going to waste your time.

Creating a repository with `jj git init`

Let's make a new repository! First, we need a project to track. I am going to use a Rust project in this example, since it is my favorite language, but you can use whatever you'd like: we won't be writing complex code here, just giving ourselves something to work with.

We can use `cargo new` to make a new Rust project, and we'll tell it not to create any version control repository for us, so we can do it ourselves.

```
$ cargo new hello-world --vcs=none
    Created binary (application) `hello-world` package
$ cd hello-world
```

In Cargo projects, the main source file is stored in `src/main.rs`, whose contents look like this:

```
fn main() {
    println!("Hello, world!");
}
```

Perfect. Now, this is kind of funny, but `jj` doesn't have an equivalent of `.gitignore`, and instead, just supports `.gitignore`. So let's put this in a `.gitignore` file:

```
/target/
```

If you're using another language, you may want to add something like `node_modules` if you're in JavaScript, or the equivalent of whatever language you're using.

Now that we've got a project, let's initialize our repository:

```
$ jj git init
Initialized repo in "."
```

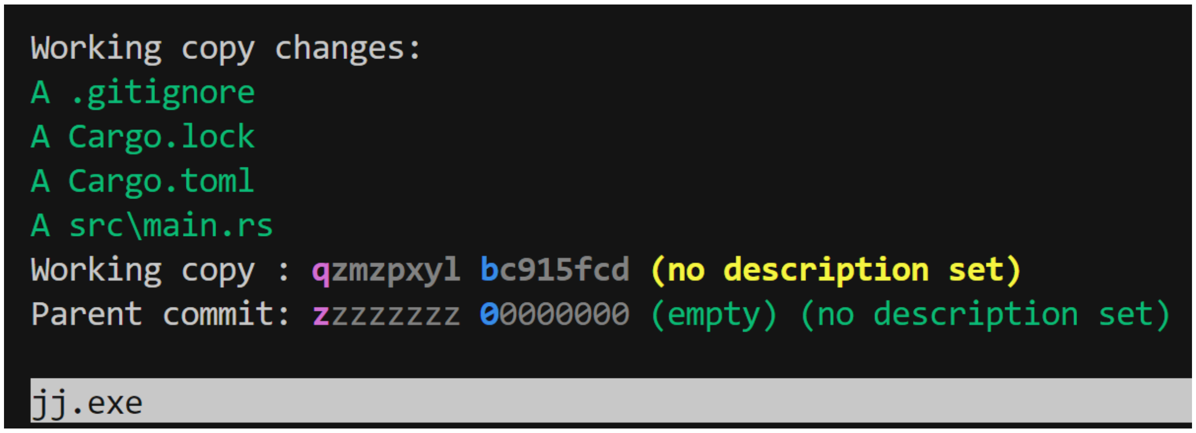
Now, you may be wondering, "why not just `jj init`?" The deal is this: the native repository format is still a work in progress. So we're creating a repository that's backed by a real `git` repository, because in practice, this early in `jj`'s life, that's the right thing to do.

Anyway, now we've got a repository! In the next section, we'll take a peek inside.

Viewing the current status with `jj st`

We can view the status of our repository with `jj st`. Let's run that now:

```
$ jj st
```



```
Working copy changes:
A .gitignore
A Cargo.lock
A Cargo.toml
A src\main.rs
Working copy : qzmzpxyl bc915fcd (no description set)
Parent commit: zzzzzzzz 00000000 (empty) (no description set)

jj.exe
```

This is the `jj` "pager", a program that shows the output of commands, and lets you scroll through them if they get really long. You can hit `q` to get back to your console.

You can request to not use the pager by using `jj st --no-pager`, or if you hate the pager and want to turn it off, you can configure that with

```
$ jj config set --user ui.paginate never
```

I am going to present the examples in the book as if this is the configuration, because managing text is easier than wrangling screenshots. The same information ends up on the screen no matter which way you prefer. Speaking of, let's actually talk about the output of `jj st`:

```
$ jj st --no-pager
Working copy changes:
A .gitignore
A Cargo.lock
A Cargo.toml
A src\main.rs
Working copy : qzmzpxyl bc915fcd (no description set)
Parent commit: zzzzzzzz 00000000 (empty) (no description set)
```

There's a surprising amount of stuff to talk about here! Let's dig into it.

```
Working copy changes:
A .gitignore
A Cargo.lock
A Cargo.toml
A src\main.rs
```

This is the first thing we need to talk about: unlike `git`, `jj` has no index. Wait, don't close the tab! Here's the thing: `jj` gives you the same ability to craft artisanal, beautiful commits that have exactly what you want in them. But it doesn't need an index to do it.

This is a running theme with `jj`: it gives you fewer tools, but those tools end up having equivalent or even more power than their `git` counterparts. Because there are fewer tools, there's also less to learn. Now I am not one of those "the `git` CLI is too complex and `git` is too hard to learn" people, but I do acknowledge that puts me in the minority. But let's reframe that: if we can make something more powerful *and* easier? Sign me up!

We'll get into how to reproduce the power of an index later. For now, what we need to know is that every time you run a `jj` command, it examines the working copy (the files on disk) and takes a snapshot. So here, it's noticed that we've `A`dded some new files. You'll also see `M`odified files, and `D`eleted files.

```
Working copy : qzmzpxyl bc915fcd (no description set)
Parent commit: zzzzzzzz 00000000 (empty) (no description set)
```

Our brand new repo shows that we have two *changes*. You'll notice that the text on the second one says "commit" there, and... yeah okay so: `jj` has a few different concepts here. The first is a commit. Our two commits have the identifiers `bc915fcd` and `00000000`. But there's also the idea of a "change," and that's that in `jj`, commits can evolve over time. But we still need a stable identifier to talk about those changes, so we have a "change ID," and that's `qzmzpxyl` and `zzzzzzzz`. One really cool thing is that they use a disjoint set of identifiers: `qzmzpxyl` can never be a commit ID, but must be a change ID, and `bc914fcd` can never be a change ID, but must be a commit ID. This is surprisingly handy.

Anyway, we'll talk more about commits and changes soon, and how they're different, but first we should talk about the rest of the details here. The first part is that each repository always has a `zzzzzzzz 00000000` change, and it's always empty. This is called the "root commit" and it is the foundation of the whole repository. Given that it's empty, `jj` has created a second change based on top of it, in this case, `qzmzpxyl` and it is tracking the contents of the working copy. Since it's not empty, its line here doesn't have the `(empty)` bit like our root change has.

Finally, both of our changes say `(no description set)`, and that's because we haven't given them a description yet! We'll talk about descriptions in the next section.

Using `jj describe` to describe our commits in a human-friendly way

While we can refer to our changes by their change ID or commit ID, that's not always great. Text is a much better way to describe things for humans.

However, before we can describe commits, we have to let `jj` know who we are. Let's set some quick configuration:

```
$ jj config set --user user.name "Steve Klabnik"
$ jj config set --user user.email "steve@steveklabnik.com"
```

Obviously, unless you're me, you should be putting your own name and email in there. Okay, with that out of the way, we're ready to describe some changes.

Whenever we feel like it, we can describe our changes with `jj describe`. The simplest way to use it is with the `-m`, or "message" flag. This allows us to pass the description on the command line:

```
$ jj describe -m "hello world"
Working copy now at: yyrsmnoo 524d2bf4 hello world
Parent commit      : zzzzzzzz 00000000 (empty) (no description set)
```

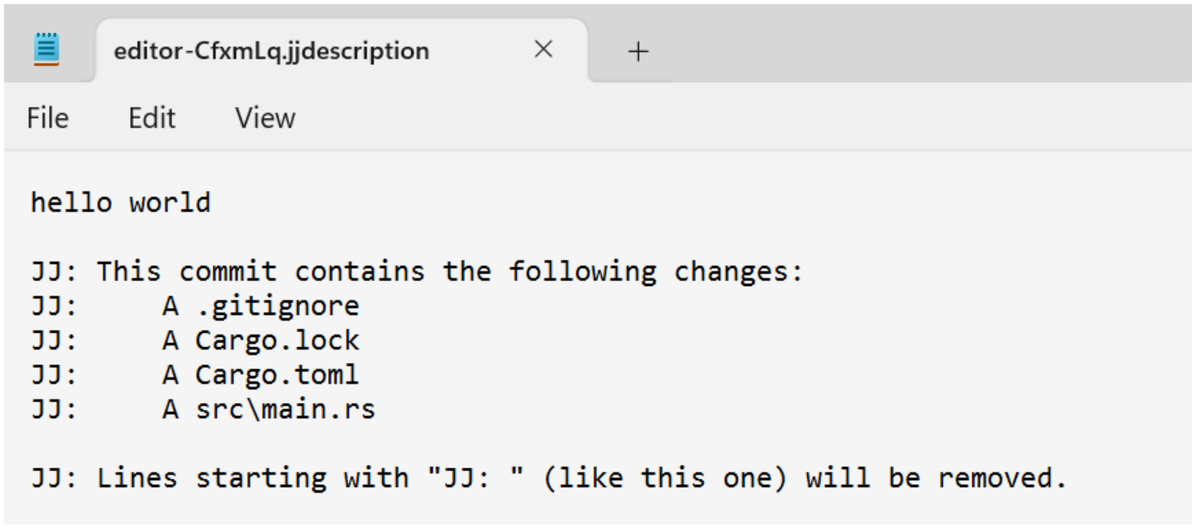
(You may notice that the change ID changed here: that's just some book-writing magic. I am editing this book manually, and so may make adjustments that end up giving you different change IDs and commit IDs than I do. You'll figure it out, just match the output of your commands to the inputs you give and you'll be fine.)

Our message, `hello world`, has replaced the `(no description set)` text. We're gonna be able to see this whenever we look at our repository history.

For more real changes though, you'll probably want to not use the `-m` flag. And, since descriptions can be set at any time, we can also change them too. Let's try it again:

```
$ jj describe
```

An editor will pop up; I'm on Windows, so I'm getting notepad.



```
hello world

JJ: This commit contains the following changes:
JJ:     A .gitignore
JJ:     A Cargo.lock
JJ:     A Cargo.toml
JJ:     A src\main.rs

JJ: Lines starting with "JJ: " (like this one) will be removed.
```

This window shows my original message, "hello world," and then a bunch of lines that start with `JJ:` . As the final one mentions, these lines are ignored when forming the commit description. So let's make a longer description, like this:

```
hello world

This is an initial "Hello, world!" implementation, nothing fancy.

More fun stuff to come.

JJ: This commit contains the following changes:
JJ:     A .gitignore
JJ:     A Cargo.lock
JJ:     A Cargo.toml
JJ:     A src\main.rs

JJ: Lines starting with "JJ: " (like this one) will be removed.
```

After saving and closing, we'll get this output:

```
Working copy now at: yyrsmnoo ac691d85 hello world
Parent commit      : zzzzzzzz 00000000 (empty) (no description set)
```

We only see that first line, but the rest are still there.

Eagle eyed readers may notice one other change. Let's take two of these outputs and put them next to each other:

```
Working copy now at: yyrsmnoo 524d2bf4 hello world
Working copy now at: yyrsmnoo ac691d85 hello world
```

Changing our description changed the commit ID! This is why we have both IDs: the change ID has not changed, but the commit ID has. This allows us to evolve our commit over time, but still have a stable way to refer to all versions of it.

We will come back to this more in the future, because first, I'd like to show you how to make new changes.

Using `jj new` to create new changes

We're done with our first commit, and we're ready to do more work. Let's start that work by using `jj new`:

```
$ jj new
Working copy now at: puomrwxl 01a35aad (empty) (no description set)
Parent commit      : yyrsmnoo ac691d85 hello world
```

It's that easy! We now have a new change, `puomrwxl`, that's empty and has no description. But its parent is our previous change, `yyrsmnoo`.

Let's check out `jj st`:

```
$ jj st
The working copy is clean
Working copy : puomrwxl 01a35aad (empty) (no description set)
Parent commit: yyrsmnoo ac691d85 hello world
```

Nice, a clean working directory: all of our changes were made in `yyrsmnoo`, and we're starting this change fresh.

We now technically have a very primitive, but near-complete, workflow. That's really all you need to know to get started. To practice, let's make another change. This time, I'm going to describe things first, before I make any changes:

```
> jj describe -m "it's important to comment our code"
Working copy now at: puomrwxl a0f0bc71 (empty) it's important to
comment our code
Parent commit      : yyrsmnoo ac691d85 hello world
```

Just what we expected, still an empty change, but with a description, and our commit ID has updated while the change ID stays the same.

Let's modify `src/main.rs`:

```
/// A "Hello, world!" program.

fn main() {
    println!("Hello, world!");
}
```

We can double check that `jj` has noticed our change:

```
$ jj st
Working copy changes:
M src\main.rs
Working copy : puomrwxl 7a096b8a it's important to comment our code
Parent commit: yyrsmnoo ac691d85 hello world
```

Excellent, `src\main.rs` has been `M` odified, we have a new commit ID. Since we're done with this change, let's start a new one:

```
$ jj new
Working copy now at: ywnkulko 46b50ed7 (empty) (no description set)
Parent commit      : puomrwxl 7a096b8a it's important to comment our
code
```

Wonderful.

Just seeing the parent change is very restrictive, though. It would be nice if we could look at all of the work we've done. Let's tackle that next.

Viewing the contents of your repository with jj log

Let's look at our chain of changes:

```
> jj log
@  ywnkulko steve@steveklabnik.com 2024-02-28 20:40:00.000 -06:00
46b50ed7
│  (empty) (no description set)
◉  puomrwxl steve@steveklabnik.com 2024-02-28 20:38:13.000 -06:00
7a096b8a
│  it's important to comment our code
◉  yyrsmnoo steve@steveklabnik.com 2024-02-28 20:24:56.000 -06:00
ac691d85
│  hello world
◉  zzzzzzzz root() 00000000
```

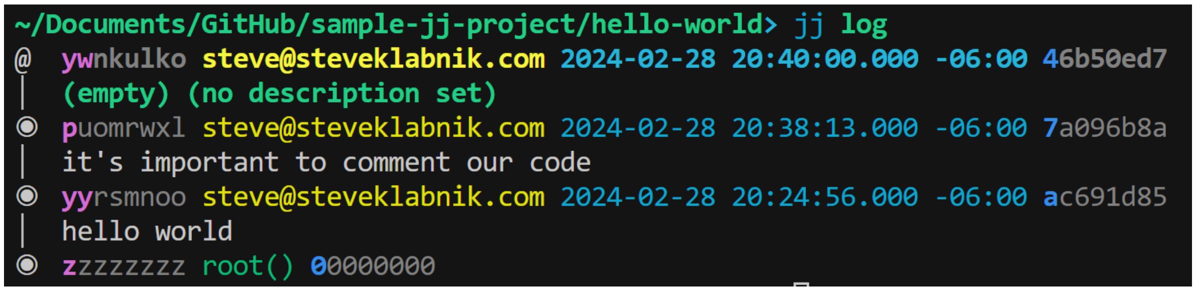
As you can see, this is sort of like `git log`, but also very different. There's a bunch going on here, let's talk about various parts of this.

The very first character at the top left is an `@`. `@` is a special name for "whichever commit the working copy reflects." At first I kind of thought about it like `HEAD` in `git`, but that's not correct: `HEAD` is the most recent commit, but `@` represents the working copy, which may be "dirty" from `git`'s point of view. This is our first glimpse into the power of the index-less workflow, though we'll explore that fully in the next chapter. For this moment, just realize that we have one less concept, but haven't actually lost any of its power.

Next, we see the change ID. On the far right, we see the commit ID. This is because when viewing things this way, we almost certainly don't care about the commit ID: we care about the sequence of stable change identifiers. Between the two is the author and the time, and on the line below, we have our description.

At the bottom, we have our root commit, but instead of an author and a time, it says `root()`. This is a *revset*, which is a feature we'll explore later. But the short of it is this: `jj` has a really powerful way to select lists of revisions. `root()` is a function in this language (yes, it has functions) that returns the root commit.

One more thing: my text representation of the output of `jj log` was missing something. Here's a screenshot of my terminal, and you may notice something interesting:



```
~/Documents/GitHub/sample-jj-project/hello-world> jj log
@  ywnkulko steve@steveklabnik.com 2024-02-28 20:40:00.000 -06:00 46b50ed7
│  (empty) (no description set)
◉  puomrwxl steve@steveklabnik.com 2024-02-28 20:38:13.000 -06:00 7a096b8a
│  it's important to comment our code
◉  yyrsmnoo steve@steveklabnik.com 2024-02-28 20:24:56.000 -06:00 ac691d85
│  hello world
◉  zzzzzzzz root() 00000000
```

The output has color, and a lot of it! But the most important bit is the highlights in the revision IDs and commit IDs. See how `yw` is in magenta, but the rest of the ID is in grey, `nkulko`? Similar to `git`, when talking about an ID, you only need to refer to the unique prefix, and not the whole ID. So that magenta bit is showing you said prefix; we could run commands that refer to `ywnkulko` or `yw` and they'd both work. Why is it two characters? Well, because we also have `yyrsmnoo`. But see how the middle change, `p` is the prefix and `uomrwxl` is in grey? Because this is the only change that starts with `p`. But since we have two changes starting with `y`, we need a second character to make them unique.

It's very cool that the UI is communicating this to us! And, it's also why the format of change IDs is pretty cool: by using letters, there are less likely to be conflicts, which means these are very often very short, until your repository grows pretty large.

There are many more secrets to `jj log`, but for now, this is enough to be able to go back and look at all of your changes, so we're gonna keep it there. Let's recap what we've learned.

A recap and some thoughts

So here is our current workflow:

1. Create new repositories with `jj git init`.
2. To start working on a new change, use `jj new`.
3. To describe a change so humans can understand them, use `jj describe`.
4. We can look at our work with `jj st`.
5. When we're done, we can start our next change with `jj new`.

Finally, we can review our repository's contents with `jj log`.

This is... pretty simple! We don't need that many concepts to get started. Of course, we aren't yet able to do some very important things like "share our code with others." But we'll get there.

An interesting thing that I've noticed while using `jj` is that sometimes things feel the same, but backwards. In `git`, we finish a set of changes to our code by committing, but in `jj` we start new work by creating a change, and *then* make changes to our code. It's more useful to write an initial description of your intended changes, and then refine it as you work, than creating a commit message after the fact.

But also, this stuff is flexible: why *should* you have to create a commit message at the time of creating a commit, and not whenever you feel like it? The same stuff exists, but in more flexible pieces that I can combine together.

Before we get into topics like "how to share code with others" and more details about some of the things we've already learned, let's talk about making our workflow a little nicer; you should get some more practice using `jj` before we start collaborating.

Real-world workflows

We can use `jj` at this point, but I wouldn't say our workflow is great. In this chapter, we're going to explore two different workflows that are popular with `jj` users. Like `git`, `jj` is extremely flexible, and so you can customize your workflow in many ways, but I'd like to show you two examples. For me, understanding the basics we've talked about wasn't tough, but when I actually sat down to *use* `jj`, I found myself tripping up a bit. With a few more commands, we can have nicer workflows that let us work a bit more naturally.

We'll start with the "squash workflow," as it is the workflow that Martin, the creator of `jj`, prefers. We'll then talk about the "edit workflow," which is popular among people who don't like the squash workflow.

The Squash Workflow

The first workflow we're going to talk about is the "squash workflow," the one preferred by `jj` 's creator, Martin. It's called "the squash workflow" because it uses a command we haven't interacted with yet, `jj squash` . The second reason that I'm talking about this workflow first is that it is the workflow that should appeal to people who are big fans of `git` 's index, and I think comparing and contrasting the two is interesting.

The workflow goes like this:

1. We describe the work we want to do.
2. We create a new empty change on top of that one.
3. As we produce work we want to put into our change, we use `jj squash` to move changes from `@` into the change where we described what to do.

In some senses, this workflow is like using the `git` index, where we have our list of current changes (in `@`), and we pull the ones we want into our commit (like `git add`).

Starting work by describing it

Let's recap where we are in our project: `@` currently is an empty commit:

```
> jj log
@  ywnkulko steve@steveklabnik.com 2024-02-28 20:40:00.000 -06:00
46b50ed7
│  (empty) (no description set)
◉  puomrwxl steve@steveklabnik.com 2024-02-28 20:38:13.000 -06:00
7a096b8a
│  it's important to comment our code
◉  yyrsmnoo steve@steveklabnik.com 2024-02-28 20:24:56.000 -06:00
ac691d85
│  hello world
◉  zzzzzzzz root() 00000000
```

Let's describe the work that we want to do:

```
$ jj describe -m "print goodbye as well as hello"
Working copy now at: ywnkulko 4bfe3940 (empty) print goodbye as well
as hello
Parent commit      : puomrwxl 7a096b8a it's important to comment our
code
```

This change is currently empty, but we've now given it a useful name. This is the change we're going to build up over time. But for now, it's empty.

Create a new empty change

We need a place to hold our changes until we decide if we want to put them in our commit or not. So let's make a new one:

```
$ jj new
Working copy now at: rkvxolny 5e020e00 (empty) (no description set)
Parent commit      : ywnkulko 4bfe3940 (empty) print goodbye as well
as hello
```

We now have our change. It's also empty! There's no issue having two empty commits, one after the other. And since we are using this like an index, we dont really need to give it a name either. It's just a scratch space, but since it's part of a change, we're "always committed" in a sense.

Now it's time for the fun stuff.

Use `jj squash` to move things into our "real" change

Let's make a change to our code:

```
/// A "Hello, world!" program.

fn main() {
    println!("Hello, world!");
    println!("Goodbye, world!");
}
```

Now that our "feature" has been implemented, let's see our current changes:

```
$ jj st
Working copy changes:
M src\main.rs
Working copy : rkvxolny aee5266d (no description set)
Parent commit: ywnkulko 4bfe3940 (empty) print goodbye as well as
hello
```

We now have an even wilder situation than before: our current change has stuff in it, but the parent is empty! Let's change that. We want to take our change from our "staging area" and put it into our commit (change). We can do that with `jj squash`:

```
$ jj squash
Working copy now at: oopolqyp 9fb63b14 (empty) (no description set)
Parent commit      : ywnkulko ed71bb54 print goodbye as well as
hello
```

Lots of changes here! `@` is now empty, with no description, and the parent is now no longer empty. Our changes are now in `ywnkulko`.

What we did is kind of the equivalent of `git commit -a --amend`. But what about more focused changes? Well, if we only want to add a specific file, like `git add <file> && git commit --amend` we can pass it as an argument. Because we only had one file, the previous command was equivalent to

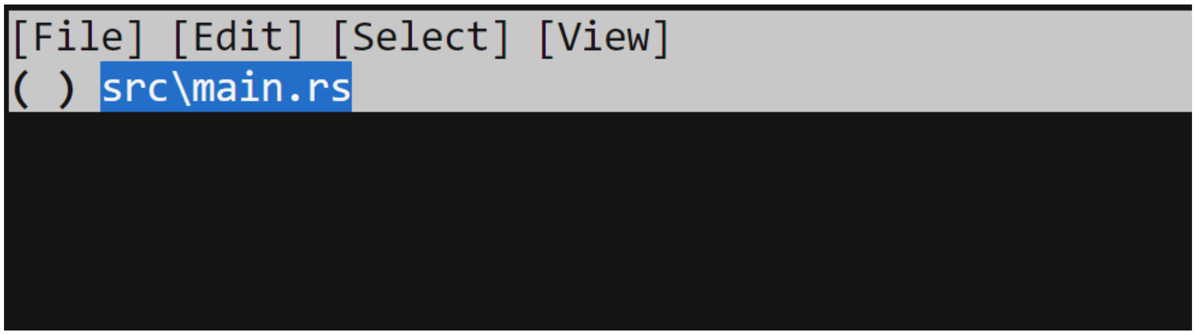
```
$ jj squash src/main.rs
```

But we can also get the equivalent of `git add -p && git commit --amend`, where we only add parts of a file to our commit. And it's gonna blow your mind.

Make a few changes around your source file, and then do this:

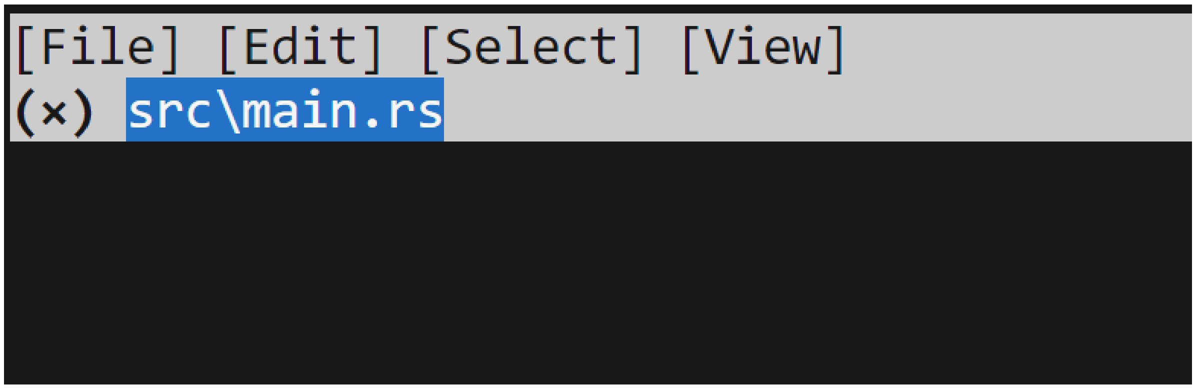
```
$ jj squash -i
```

This will bring up a TUI!



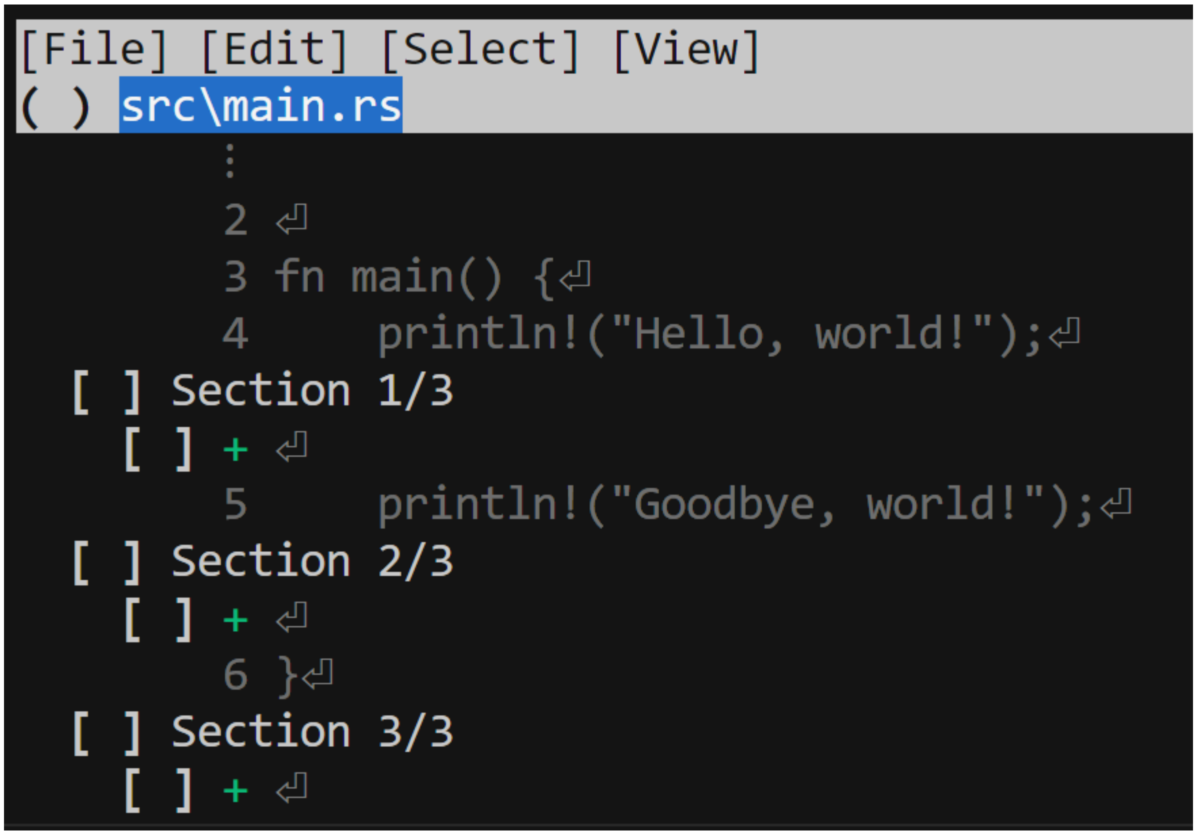
```
[File] [Edit] [Select] [View]
( ) src\main.rs
```

By default, it's showing a file-level view: we have our one file, and the `( )` indicates that we haven't selected to include this. We could do so by hitting space, and that will fill the parenthesis in with a ● (screenshot outdated and shows an x):



```
[File] [Edit] [Select] [View]
(×) src\main.rs
```

Let's press space again to undo that, and then hit `f` to toggle "folding":



```
[File] [Edit] [Select] [View]
( ) src\main.rs
        ⋮
        2 ⏎
        3 fn main() {⏎
        4     println!("Hello, world!");⏎
  [ ] Section 1/3
    [ ] + ⏎
        5     println!("Goodbye, world!");⏎
  [ ] Section 2/3
    [ ] + ⏎
        6 }⏎
  [ ] Section 3/3
    [ ] + ⏎
```

I added empty spaces in a few places, and you can see each individual section and line has its own checkbox. We can use the mouse to click, or arrow keys to navigate, and space to toggle if we're accepting these changes.

Once we're done, we can hit `c` to confirm our changes. I'm not selecting any, since these were just nonsense stuff I wanted to add to show off the TUI. I dont want to keep them at all, so let's just dump them. We can get rid of the stuff in `@` with `jj abandon`:

```
$ jj abandon
Abandoned commit oopolqyp 44665581 (no description set)
Working copy now at: ootnlvpt 97b7a559 (empty) (no description set)
Parent commit      : ywnkulko ed71bb54 print goodbye as well as
hello
Added 0 files, modified 1 files, removed 0 files
```

We've thrown away `oopolqyp`, and `jj` has helpfully made a new empty change for us.

This is the kind of stuff I mean when I say "the same power, but less concepts." We've got the tools that the index gave us, but they're simpler because we don't use some of them on the index, and some on commits: we use them all on commits.

That also implies that "the same power" isn't exactly true: `jj squash` is more powerful than `git add` because it can work on *any* change and its parent, moving stuff between them. This gets into the kind of shenanigans you can get up to with `git rebase -i`, without the need for *another* command like that. Simpler, but *more* powerful, thanks to orthogonality.

Recap and thoughts

This workflow is not super different than our previous one, but by adding one more command, we get a bit more power. And we've learned that we can use `jj squash` to move contents of changes into their parent.

The Edit Workflow

While I like the previous workflow, some people just don't. They use a workflow that adds a different command, `jj edit`, along with a second new command, `jj next`, as well as a new flag to `jj new`. Lots to learn!

The workflow goes like this:

1. We create a new change to work on our feature.
2. If we end up doing exactly what we wanted to do, we're done.
3. If we realize we want to break this up into smaller changes, we do it by making a new change before the current one, swapping to it, and making that change.
4. We then go back to the main change.

Let's see how to use `jj` this way.

Create a new change to work on our feature.

Let's create a feature that's un-doing our previous feature: we'll revert to `Hello, World!` only.

Now, our previous workflow left `@` at an empty change. But if you use this workflow, `@` will often be on an existing change. So in the real use of this workflow, we'd start by:

```
$ jj new -m "only print hello world"
```

But since we have an empty change, what we'll actually do is:

```
> jj describe -m "only print hello world"
Working copy now at: ootnlvpt bb06f041 (empty) only print hello
world
Parent commit      : ywnkulko ed71bb54 print goodbye as well as
hello
```

We are now ready to do some work.

Let's change our file to:

```
/// A "Hello, world!" program.

fn main() {
    println!("Hello, world!");
}
```

Cool. We're done. In the best case, we're happy with this change, and we're done. When we begin more work we start it with `jj new -m ""` and get to work.

But sometimes, when we're working on something, we realize we also want a different change, and maybe it relies on this one. For example, let's say that we were working on undoing this goodbye feature, but we realized we wanted to refactor printing out into its own function, because that's a terrible idea in practice and so makes for a good example to play around with.

What we want to do is make a new change before this one. So let's do that.

Make a new change and edit it

Let's try this:

```
$ jj new -B @ -m "add more comments"
Rebased 1 descendant commits
Working copy now at: nmptruqn 30a1f33b (empty) add more comments
Parent commit      : ywnkulko ed71bb54 print goodbye as well as
hello
Added 0 files, modified 1 files, removed 0 files
```

We have a new flag to `jj new`, `-B`. This says to create the new change *before* the current one. That's exactly what we asked!

The first line of the output should raise some eyebrows:

```
Rebased 1 descendant commits
```

That's right, because we have created a change before the one we're on, it automatically rebased our original change. How can it do that? What if there are conflicts? Relax, we'll get there. All I'll say is something that's probably hard to believe: this operation will *always* succeed, and we will have our working copy at the commit we've just inserted. You won't learn how this works in this chapter, but in a future one.

In the meantime, let's examine our log:

```
$ jj log
◉  ootnlvpt steve@steveklabnik.com 2024-02-28 22:59:46.000 -06:00
be40656e
│  only print hello world
@  nmptruqn steve@steveklabnik.com 2024-02-28 22:59:46.000 -06:00
30a1f33b
│  (empty) add more comments
◉  ywnkulko steve@steveklabnik.com 2024-02-28 22:09:40.000 -06:00
ed71bb54
│  print goodbye as well as hello
◉  puomrwxl steve@steveklabnik.com 2024-02-28 20:38:13.000 -06:00
7a096b8a
│  it's important to comment our code
◉  yyrsmnoo steve@steveklabnik.com 2024-02-28 20:24:56.000 -06:00
ac691d85
│  hello world
◉  zzzzzzzz root() 00000000
```

We can see that `@` is at our new empty change, and that we have our original change, `ootnlvpt`, is after us. Some of you may recognize `ootnlvpt`: even though we rebased it on top of our current change, `nmptruqn`, the change ID is the same. The commit changed from `bb06f041` to `be40656e`, though. The change ID is stable, but we can keep track of how the commit changes over time. Neat.

Anyway, now we can edit `@`. Let's change `src/main.rs`. When you first open up the file, you'll see this:

```
/// A "Hello, world!" program.

fn main() {
    println!("Hello, world!");
    println!("Goodbye, world!");
}
```

Remember, this change is before the one where we removed the goodbye message, so that has returned. Here's what we want to end up with:

```
/// A "Hello, world!" program.
///
/// This is the best implementation of this program to ever exist.

fn main() {
    println!("Hello, world!");
    println!("Goodbye, world!");
}
```

This is very silly. Regardless, we have finished. Let's see our current status:

```
$ jj st
Rebased 1 descendant commits onto updated working copy
Working copy changes:
M src\main.rs
Working copy : nmptruqn 90a2e97f add more comments
Parent commit: ywnkulko ed71bb54 print goodbye as well as hello
```

Yet again, a rebase. Because we have changed the contents of our change, all of the changes that depend on it must be rebased. But again, this happens all the time, without fail. So it's not something you'll get stuck on at this stage.

Return to our main change

Now that we're done, we're going to go back to editing our original commit. To do that, we could use `jj edit`, which is where this workflow gets its name from. `jj edit` sets the working copy to the contents of a change, and now changes you make will update that change.

Doing that would look like this:

```
$ jj edit o
```

Since `o` is the unique prefix of `ootnlvpt`, our original feature change. However, looking up that revision is kind of annoying. Therefore, we can use a simpler command:

```
$ jj next --edit
Working copy now at: ootnlvpt e13b2585 only print hello world
Parent commit      : nmptruqn 90a2e97f refactor printing
Added 0 files, modified 1 files, removed 0 files
```

`jj next` will move `@`, the working copy change, to the child of where it is now. The `--edit` flag means we're now going to be editing that change, whereas if you leave it off, it works more like a variant of `jj new`, making a new change based on top of that change.

Let's double check with `jj log`:

```
$ jj log
@  ootnlvpt steve@steveklabnik.com 2024-02-28 23:26:44.000 -06:00
b5db7940
│  only print hello world
◉  nmptruqn steve@steveklabnik.com 2024-02-28 23:09:11.000 -06:00
90a2e97f
│  add more comments
◉  ywnkulko steve@steveklabnik.com 2024-02-28 22:09:40.000 -06:00
ed71bb54
│  print goodbye as well as hello
◉  puomrwxl steve@steveklabnik.com 2024-02-28 20:38:13.000 -06:00
7a096b8a
│  it's important to comment our code
◉  yyrsmnoo steve@steveklabnik.com 2024-02-28 20:24:56.000 -06:00
ac691d85
│  hello world
◉  zzzzzzzz root() 00000000
```

That's correct, `@` is at our original change.

Recap and thoughts

This workflow is also a good alternative. If your brain thinks this way is better than the other way, that's great! A nice thing about the flexibility of these tools is you can work with them how you'd like!

Branching, merging, and conflicts

You may have noticed that we haven't talked about branches at all yet. This is another significant difference between `jj` and `git`: `jj` prefers to use anonymous branches, rather than named ones. People sometimes call this a "branchless" workflow. In this chapter, we'll learn about how to do branches in the way `jj` prefers, and in the next chapter, we'll talk about named branches.

Here's what we're going to learn:

- What anonymous branches are, and how to use them
- Figuring out where our changes are with revsets
- Merging anonymous branches
- Dealing with conflicts

What anonymous branches are, and how to use them

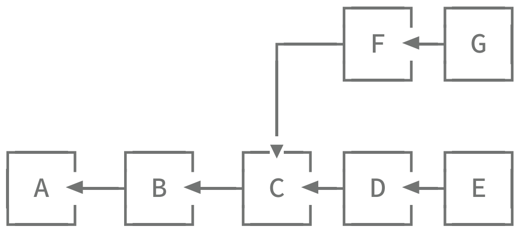
When I first heard of "anonymous branches," I got very confused. Git models branches as a pointer to a commit, and that pointer needs a name, so that's a branch.

Turns out, you don't really need to name your branches, and doing so is also not really worth it. I have heard that, inside of Meta, where they use a similar VCS tool that also has anonymous branches, almost nobody bothers to name their branches once they get used to things.

Let's talk about it.

What is a branch, conceptually?

When two changes share the same parent change, we say that they are "branching," because the graph of commits would look like this:



Here, we'd say that `F` and `G` are two changes that are "on a branch," because it looks like they're branching off from `D` and `E`. In reality, in `git`, both would be "on a branch," because everything is on some sort of branch in `git`. If you're not on a branch, `git` will say something like this:

> You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch.

Git considers any commit that's not part of a branch to be garbage, and so will garbage collect those commits at some point. Git is very branch-centric.

`jj` does not think about the world this way. Consider the diagram above: we didn't name any of these branches, yet the diagram still made sense. There's not really an inherent need to name our branches, just like there isn't an inherent need to describe our changes. That said, it would be nice to know how to refer to different branches, even if they're not named.

Let's see how this works.

Creating two branches from the same commit

Run `jj st`, and make sure your working copy is at an empty change. If not, use `jj new` to create one. We want to start from a blank slate here.

```
$ jj new
Working copy now at: yykpmnuq 0bc7a425 (empty) (no description set)
Parent commit      : ootnlvpt b5db7940 only print hello world
```

A common reason for branching is to work on two different ideas at once. Let's start two different features: one to add some more documentation to our project, and another to split our print function into hello and goodbye functions.

First, we'll work on the documentation, so let's describe our current commit to be working on that:

```
> jj describe -m "add better documentation"
Working copy now at: yykpmnuq 4a95c1f9 (empty) add better
documentation
Parent commit      : ootnlvpt b5db7940 only print hello world
```

Next, we want to make a change to work on our hello and goodbye functions. We want this change to build on top of `ootnlvpt`, so we can just say that:

```
$ jj new o
Working copy now at: xrslwzvq e9249c85 (empty) (no description set)
Parent commit      : ootnlvpt b5db7940 only print hello world
```

We want to create our new change with the parent `o`, which we could see is the short name for `ootnlvpt`. Your change ID may be different if you're not following me exactly, so you may want to double check you've got the right change!

Let's describe this one too:

```
$ jj describe -m "create hello and goodbye functions"
Working copy now at: xrslwzvq a70d464c (empty) create hello and
goodbye functions
Parent commit      : ootnlvpt b5db7940 only print hello world
```

Excellent. We've got two different changes, `yykpmnuq` and `xrslwzvq`, both with the parent `ootnlvpt`. Success! We have created a branch. And we didn't need to name it.

Let's edit `src/main.rs` to update this description:

```
/// A "Hello, world!" program.
///
/// This is the best implementation of this program to ever exist.

fn main() {
    print_hello();
    print_goodbye();
}

fn print_hello() {
    println!("Hello, world!");
}

fn print_goodbye() {
    println!("Goodbye, world!");
}
```

This is pretty silly, but we'll use it further along in the tutorial.

We can see that there's a branch in the output of `jj log`:

```
$ jj log
@  xrslwzvq steve@steveklabnik.com 2024-02-29 23:06:23.000 -06:00
a70d464c
│  (empty) create hello and goodbye functions
│ ◉  yykpmnuq steve@steveklabnik.com 2024-02-29 23:03:22.000 -06:00
210283e8
├─╯  (empty) add better documentation
◉  ootnlvpt steve@steveklabnik.com 2024-02-28 23:26:44.000 -06:00
b5db7940
│  only print hello world
◉  nmptruqn steve@steveklabnik.com 2024-02-28 23:09:11.000 -06:00
90a2e97f
│  refactor printing
◉  ywnkulko steve@steveklabnik.com 2024-02-28 22:09:40.000 -06:00
ed71bb54
│  print goodbye as well as hello
◉  puomrwxl steve@steveklabnik.com 2024-02-28 20:38:13.000 -06:00
7a096b8a
│  it's important to comment our code
◉  yyrsmnoo steve@steveklabnik.com 2024-02-28 20:24:56.000 -06:00
ac691d85
│  hello world
◉  zzzzzzzz root() 00000000
```

We've got our two changes, and there's a fork in the road.

So if these branches don't have names, how do we tell them apart? Well, the descriptions of their commits are right there. We can look at them and easily tell which of the two we care about, and then use their change IDs to distinguish between the two. Coming up with an extra name isn't inherently helpful here. Of course, *sometimes* it can be, and that's why eventually we'll talk about named branches. But the important thing to realize here is that you only have to name branches where adding a name adds some sort of value.

Getting a list of branches

Okay, but what if we had tons of branches? How would we be able to see them? In this view, with only two, it makes sense, but what if we had way more?

We can actually ask `jj log` to show us the head of every anonymous branch. We do it like this:

```
> jj log -r 'heads(all())'
@  xrslwzvq steve@steveklabnik.com 2024-02-29 23:06:23.000 -06:00
a70d464c
│  (empty) create hello and goodbye functions
~

◉  yykpmnuq steve@steveklabnik.com 2024-02-29 23:03:22.000 -06:00
210283e8
│  (empty) add better documentation
~
```

This shows both of our heads. But what is that `heads(all())` stuff? It's called a "revset," and it's what we're going to talk about next.

Figuring out where our changes are with revsets

We have learned about two kinds of identifiers in `jj`: change IDs and commit IDs. But what if we want to talk about, for example, a range of commits?

`jj` has a concept called a "revset," short for "revision set." Sometimes people say "revision" instead of "commit," and "revset" is just nicer to say than "comset", so it stuck.

This sounds pretty intense at first, but I promise it's simpler than you think: `jj` supports a functional language to describe revsets. Almost every command in `jj` takes a `-r` / `--revision` flag, which is the revision to operate on. This defaults to `@`. This means when we do `jj new`, we're basically doing `jj new -r @`, that is, create a new change with a parent revision of the current working copy.

Symbols

`@` is actually our first example of the revset language. This is called a "symbol". Symbols are a means of specifying a single commit. `@` refers to the change containing the current working copy, but a change ID or commit ID are other examples of symbols.

Operators

Operators let you describe more complex relationships between changes. For example, remember how in the squash workflow, we would move the contents of the working directory into the parent change? Well, the `-` operator refers to the parent of a given revision, and `@` is the change referring to the current working directory, so we might say "we squashed the contents of `@` into `@-`.

And in fact, `jj squash` is short for `jj squash -r @`. There are many operators, including, but not limited to:

- `x & y`: changes that are in both x and y
- `x | y`: changes that are in either x or y
- `::x` Ancestors of x
- `x::` Descendants of x

And more. The final bit is the most interesting, and that's functions.

Functions

Functions allow for even more complex selection of a series of changes. The simplest functions are:

- `root()`: a function that returns the root change
- `all()`: this function returns all visible changes
- `mine()`: this function returns all changes authored by the current user

More complex functions can take arguments:

- `parents(x)`: the parent changes of `x`
- `ancestors(x)`: the same as `::x`, but see the next example
- `ancestors(x, depth)`: limits the results to a certain depth, which you can't do with the `::x` syntax
- `heads(x)`: commits in `x` that are not ancestors of other commits in `x`.
- `description(x)`: commits that have `x` in their description

Putting it all together

Now we can understand `heads(all())` from before: these are two functions, where we're asking for the head commits of every commit in the repository.

Revsets are very powerful, and very convenient. Would you like to find every commit by me containing the word "print" in the description? Try this:

```
$ jj log -r 'author("Steve Klabnik") & description(print)'
```

Another really useful revset function is `trunk()`:

```
$ jj log -r 'trunk()'
◉  zzzzzzzz root() 00000000
```

Right now, this doesn't look very useful, but it will be more useful when we get into sharing our changes. `trunk()` looks for a remote named `origin` or `upstream`, and looks for a `main`, `master`, or `trunk` branch, and then provides that. Since we don't have any of those right now, it gives us the same as `root()`.

Additionally, on the `jj` Discord, several folks have settled on this as a decent revset for larger repositories:

```
$ jj log -r '@ | ancestors(remote_bookmarks().., 2) | trunk()'
```

This will show the history from the working directory, some detail about remote branches, as well as the trunk. What's good varies between what you're trying to do and what your repository looks like, so experiment with some of this stuff to find something that works well for you.

Revsets are very powerful, and you'll learn some useful ones as you explore more. At some point, we'll even talk about how to create custom aliases for revsets, but for now, let's get back to dealing with branches and how to merge them.

Merging anonymous branches

Let's recall where we are:

```
> jj log
@  xrslwzvq steve@steveklabnik.com 2024-02-29 23:06:23.000 -06:00
a70d464c
│   (empty) create hello and goodbye functions
│ ◉  yykpmnuq steve@steveklabnik.com 2024-02-29 23:03:22.000 -06:00
210283e8
├─╯  (empty) add better documentation
◉  ootnlvpt steve@steveklabnik.com 2024-02-28 23:26:44.000 -06:00
b5db7940
│  only print hello world
```

We have what we consider to be the head of our repository, `ootnlvpt`, and then two branches, `xrslwzvq` and `yykpmnuq`. Let's create another change with `ootnlvpt` as the parent, to simulate the idea that some changes have landed on our main branch while we were doing the work:

```
> jj new o -m "added some cool new feature"
Working copy now at: pzoqtwuv 9353442b (empty) added some cool new
feature
Parent commit      : ootnlvpt b5db7940 only print hello world
```

Let's take a look:

```
> jj log --limit 5
@  pzoqtwuv steve@steveklabnik.com 2024-03-01 15:06:59.000 -06:00
9353442b
│  (empty) added some cool new feature
│ ◉  xrslwzvq steve@steveklabnik.com 2024-02-29 23:06:23.000 -06:00
a70d464c
├─╯  (empty) create hello and goodbye functions
│ ◉  yykpmnuq steve@steveklabnik.com 2024-02-29 23:03:22.000 -06:00
210283e8
├─╯  (empty) add better documentation
◉  ootnlvpt steve@steveklabnik.com 2024-02-28 23:26:44.000 -06:00
b5db7940
│  only print hello world
◉  nmptruqn steve@steveklabnik.com 2024-02-28 23:09:11.000 -06:00
90a2e97f
│  refactor printing
```

We passed the `--limit` flag so that we didn't see every commit; our history is already getting a little long.

Merging branches

Now, you may expect that you'd use a command like `jj merge` to merge branches together. However, as of `0.14.0`, `jj merge` is deprecated, and will be removed some time later this year. So how the heck do we create merges?

Well, what is a merge anyway? It's a new change that has more than one parent. How do we make new changes? With `jj new`. So let's ask it to make a change that has both `pzoqtwuv` and `yykpmnuq` as parents:

```
> jj new pzoqtwuv yykpmnuq -m "merge better documentation"
Working copy now at: rxzyvnkx f1c1bde8 (empty) merge better
documentation
Parent commit      : pzoqtwuv 9353442b (empty) added some cool new
feature
Parent commit      : yykpmnuq 210283e8 (empty) add better
documentation
```

Just like we'd pass a parent revision to `jj new`, we can pass multiple parents, and it just works. No need for a special command. Let's look at our history, choosing six as the limit since we just added a new change:

```
> jj log --limit 6
@    rxzyvnkx steve@steveklabnik.com 2024-03-01 15:21:11.000 -06:00
f1c1bde8
├─╮   (empty) merge better documentation
│ ◉  yykpmnuq steve@steveklabnik.com 2024-02-29 23:03:22.000 -06:00
210283e8
│ │   (empty) add better documentation
◉ │  pzoqtwuv steve@steveklabnik.com 2024-03-01 15:06:59.000 -06:00
9353442b
├─╯   (empty) added some cool new feature
│ ◉  xrslwzvq steve@steveklabnik.com 2024-02-29 23:06:23.000 -06:00
a70d464c
├─╯   (empty) create hello and goodbye functions
◉  ootnlvpt steve@steveklabnik.com 2024-02-28 23:26:44.000 -06:00
b5db7940
│  only print hello world
◉  nmptruqn steve@steveklabnik.com 2024-02-28 23:09:11.000 -06:00
90a2e97f
│  refactor printing
```

We can see the lines connecting to both of our parents, and we can still see the `xrslwzvq` branch is left over too.

But here's something really wild: we can just do this as much as we want, no need to stop at two parents. To try this out, we're going to run a command I haven't told you about yet:

```
$ jj undo
Working copy now at: pzoqtwuv 9353442b (empty) added some cool new
feature
Parent commit      : ootnlvpt b5db7940 only print hello world
$ jj log --limit 5
@  pzoqtwuv steve@steveklabnik.com 2024-03-01 15:06:59.000 -06:00
9353442b
│  (empty) added some cool new feature
│ ◉  xrslwzvq steve@steveklabnik.com 2024-02-29 23:06:23.000 -06:00
a70d464c
├─╯  (empty) create hello and goodbye functions
│ ◉  yykpmnuq steve@steveklabnik.com 2024-02-29 23:03:22.000 -06:00
210283e8
├─╯  (empty) add better documentation
◉  ootnlvpt steve@steveklabnik.com 2024-02-28 23:26:44.000 -06:00
b5db7940
│  only print hello world
◉  nmptruqn steve@steveklabnik.com 2024-02-28 23:09:11.000 -06:00
90a2e97f
│  refactor printing
```

That's right, we can undo our last command with a simple `jj undo`. We'll talk about it more in the future. But for now, it's like our merge never happened.

Let's try merging both in at the same time:

```
$ jj new pzoqtwuv yykpmnuq xrslwzvq -m "merge two branches"
Working copy now at: vuztuxmz 717232df (empty) merge two branches
Parent commit      : pzoqtwuv 9353442b (empty) added some cool new
feature
Parent commit      : yykpmnuq 210283e8 (empty) add better
documentation
Parent commit      : xrslwzvq a70d464c (empty) create hello and
goodbye functions
$ jj log --limit 6
@      vuztuxmz steve@steveklabnik.com 2024-03-01 15:38:49.000
-06:00 717232df
├─┬─╮   (empty) merge two branches
│ │ ◉   xrslwzvq steve@steveklabnik.com 2024-02-29 23:06:23.000
-06:00 a70d464c
│ │ │   (empty) create hello and goodbye functions
│ ◉ │   yykpmnuq steve@steveklabnik.com 2024-02-29 23:03:22.000
-06:00 210283e8
│ ├─╯   (empty) add better documentation
◉ │   pzoqtwuv steve@steveklabnik.com 2024-03-01 15:06:59.000 -06:00
9353442b
├─╯   (empty) added some cool new feature
◉  ootnlvpt steve@steveklabnik.com 2024-02-28 23:26:44.000 -06:00
b5db7940
│  only print hello world
◉  nmptruqn steve@steveklabnik.com 2024-02-28 23:09:11.000 -06:00
90a2e97f
│  refactor printing
```

Just as easy as that: a merge commit with three different parents.

Once again I am reminded of the theme I discussed at the start: simpler can also be more powerful. `jj` has eliminated the need for an entire command but not lost any functionality.

But what if we didn't want to create a merge commit? Don't worry, `jj` has rebase as well.

Rebasing branches

Like `git`, `jj` has a command called `rebase`. It does what it says, it takes a change and, instead of its current "base," aka parent, moves it to have a different parent, "basing" it again, or "re-basing" it.

Let's undo our merge again:

```
$ jj undo
Working copy now at: pzoqtwuv 9353442b (empty) added some cool new
feature
Parent commit      : ootnlvpt b5db7940 only print hello world
$ jj log --limit 5
@  pzoqtwuv steve@steveklabnik.com 2024-03-01 15:06:59.000 -06:00
9353442b
│   (empty) added some cool new feature
│ ◉  xrslwzvq steve@steveklabnik.com 2024-02-29 23:06:23.000 -06:00
a70d464c
├─╯  (empty) create hello and goodbye functions
│ ◉  yykpmnuq steve@steveklabnik.com 2024-02-29 23:03:22.000 -06:00
210283e8
├─╯  (empty) add better documentation
◉  ootnlvpt steve@steveklabnik.com 2024-02-28 23:26:44.000 -06:00
b5db7940
│  only print hello world
◉  nmptruqn steve@steveklabnik.com 2024-02-28 23:09:11.000 -06:00
90a2e97f
│  refactor printing
```

Excellent. Let's keep a linear history by using a rebase instead of a merge.

You can use `jj rebase` in a few different ways. Let's show off the simplest: rebasing a single change. Let's rebase our "create hello and goodbye functions" change on top of our current change:

```
$ jj rebase -r xrslwzvq -d pzoqtwuv
```

This rebases a single revision with `-r`, to a certain destination revision, hence `-d`. Since our branch only had one revision, this would be the same as passing `-b xrslwzvq`, which would move the whole branch that revision is on, or `-s xrslwzvq`, which rebases that revision as well as all of its descendants.

We didn't get any output though. Let's look at our log:

```
$ jj log --limit 5
◉  xrslwzvq steve@steveklabnik.com 2024-03-01 16:08:37.000 -06:00
6c4afc8f
│  (empty) create hello and goodbye functions
@  pzoqtwuv steve@steveklabnik.com 2024-03-01 15:06:59.000 -06:00
9353442b
│  (empty) added some cool new feature
│ ◉  yykpmnuq steve@steveklabnik.com 2024-02-29 23:03:22.000 -06:00
210283e8
├─╯  (empty) add better documentation
◉  ootnlvpt steve@steveklabnik.com 2024-02-28 23:26:44.000 -06:00
b5db7940
│  only print hello world
◉  nmptruqn steve@steveklabnik.com 2024-02-28 23:09:11.000 -06:00
90a2e97f
│  refactor printing
```

We have rebased our commit successfully. But you may have noticed something surprising: `@` is still at `pzoqtwuv`. This actually belies a very deep difference between `jj` and `git` that I learned from Austin Seipp, one of `jj`'s maintainers. And here it is:

`jj` commands primarily operate on the data structures stored in its repository, rather than on the working copy.

This simple statement has some profound implications. One of the simplest consequences of this is `jj`'s speed. Because the working copy is itself a commit, and commits are in the database, it can treat it like any other commit. In this case, the difference is even larger: `git rebase` works on the working copy. This is why it has to stop you and make you resolve things in the case where a conflict happens, because it's about to create a new commit from the working copy, and if that's in conflict, it has to be fixed or the next commit is nonsense. We'll talk about how `jj` handles conflicts shortly, but as I said before: rebases *always* succeed in `jj`. So this change is quick: it's only modifying information in the repository, not touching any of the files we have in our working directory. This also means our working directory hasn't changed, so `@` is in the same place it was before the rebase.

Some commands do move `@` by default, like `jj new`. This is because if you're creating a new change, you probably want to start working on it. But it has a flag you can pass instead to create a new change but not modify `@`:

```
$ jj new -m "not gonna start this yet" --no-edit
Created new commit owlpoptm df6620cb (empty) not gonna start this
yet
$ jj log --limit 6
◉  owlpoptm steve@steveklabnik.com 2024-03-01 16:28:54.000 -06:00
df6620cb
│  (empty) not gonna start this yet
│ ◉  xrslwzvq steve@steveklabnik.com 2024-03-01 16:08:37.000 -06:00
6c4afc8f
├─╯  (empty) create hello and goodbye functions
@  pzoqtwuv steve@steveklabnik.com 2024-03-01 15:06:59.000 -06:00
9353442b
│  (empty) added some cool new feature
│ ◉  yykpmnuq steve@steveklabnik.com 2024-02-29 23:03:22.000 -06:00
210283e8
├─╯  (empty) add better documentation
◉  ootnlvpt steve@steveklabnik.com 2024-02-28 23:26:44.000 -06:00
b5db7940
│  only print hello world
◉  nmptruqn steve@steveklabnik.com 2024-02-28 23:09:11.000 -06:00
90a2e97f
│  refactor printing
```

New change, yet we're still where we are. Let's undo that real quick:

```
$ jj undo
$ jj log --limit 5
◉  xrslwzvq steve@steveklabnik.com 2024-03-01 16:08:37.000 -06:00
6c4afc8f
│  (empty) create hello and goodbye functions
@  pzoqtwuv steve@steveklabnik.com 2024-03-01 15:06:59.000 -06:00
9353442b
│  (empty) added some cool new feature
│ ◉  yykpmnuq steve@steveklabnik.com 2024-02-29 23:03:22.000 -06:00
210283e8
├─╯  (empty) add better documentation
◉  ootnlvpt steve@steveklabnik.com 2024-02-28 23:26:44.000 -06:00
b5db7940
│  only print hello world
◉  nmptruqn steve@steveklabnik.com 2024-02-28 23:09:11.000 -06:00
90a2e97f
│  refactor printing
```

Cool. Okay so that theory sounds cool, and it's nice that it makes things fast, but what about when we *do* want to move `@`? Well, the fact that the `--no-edit` flag is what we passed to `jj new` gave it away:

```
$ jj edit xrslwzvq
Working copy now at: xrslwzvq 6c4afc8f (empty) create hello and
goodbye functions
Parent commit      : pzoqtwuv 9353442b (empty) added some cool new
feature
$ jj log --limit 5
@  xrslwzvq steve@steveklabnik.com 2024-03-01 16:08:37.000 -06:00
6c4afc8f
│  (empty) create hello and goodbye functions
◉  pzoqtwuv steve@steveklabnik.com 2024-03-01 15:06:59.000 -06:00
9353442b
│  (empty) added some cool new feature
│ ◉  yykpmnuq steve@steveklabnik.com 2024-02-29 23:03:22.000 -06:00
210283e8
├─╯  (empty) add better documentation
◉  ootnlvpt steve@steveklabnik.com 2024-02-28 23:26:44.000 -06:00
b5db7940
│  only print hello world
◉  nmptruqn steve@steveklabnik.com 2024-02-28 23:09:11.000 -06:00
90a2e97f
│  refactor printing
```

We can now rebase our other change on top too:

```
$ jj rebase -r yykpmnuq -d xrslwzvq
$ jj log --limit 5
◉  yykpmnuq steve@steveklabnik.com 2024-03-01 16:35:47.000 -06:00
7bea29b6
│  (empty) add better documentation
@  xrslwzvq steve@steveklabnik.com 2024-03-01 16:08:37.000 -06:00
6c4afc8f
│  (empty) create hello and goodbye functions
◉  pzoqtwuv steve@steveklabnik.com 2024-03-01 15:06:59.000 -06:00
9353442b
│  (empty) added some cool new feature
◉  ootnlvpt steve@steveklabnik.com 2024-02-28 23:26:44.000 -06:00
b5db7940
│  only print hello world
◉  nmptruqn steve@steveklabnik.com 2024-02-28 23:09:11.000 -06:00
90a2e97f
│  refactor printing
```

Excellent. But before we move `@`, I want to show you a little trick. We could type in the change ID, and in this case, `yyk` is the unique prefix, so it isn't that hard. But we can also use a revset:

```
$ jj edit @+
Working copy now at: yykpmnuq 7bea29b6 (empty) add better
documentation
Parent commit      : xrslwzvq 6c4afc8f (empty) create hello and
goodbye functions
```

`+` means "the child of this revision", so `@+` is "the child revision of the working copy", which in this case is exactly where we wanted to go.

We've alluded to conflicts several times in this tutorial. We're finally ready to address those.

Dealing with conflicts

When you're merging or rebasing, if your changes are incompatible with each other, you may introduce a conflict. Conflicts are often regarded as painful by users of version control systems. `jj` can't make that pain go away entirely, but it can help a lot.

Let's deliberately introduce a conflict. First, we make a new change:

```
$ jj new -m "remove goodbye message"
Working copy now at: povouosx e2c9628c (empty) remove goodbye
message
Parent commit      : yykpmnuq 2b93da0c (empty) add better
documentation
```

And then update `src/main.rs` appropriately:

```
/// A "Hello, world!" program.
///
/// This is the best implementation of this program to ever exist.

fn main() {
    print_hello();
}

fn print_hello() {
    println!("Hello, world!");
}
```

Let's also make a new change off of the previous head:

```
$ jj new yykpmnuq -m "refactor printing"
Working copy now at: vvmrvwuz 44205653 (empty) refactor printing
Parent commit      : yykpmnuq 2b93da0c (empty) add better
documentation
Added 0 files, modified 1 files, removed 0 files
```

And if we open `src/main.rs` again, we'll see that of course, it's back to the state it was before we made our other change:

```
/// A "Hello, world!" program.
///
/// This is the best implementation of this program to ever exist.

fn main() {
    print_hello();
    print_goodbye();
}

fn print_hello() {
    println!("Hello, world!");
}

fn print_goodbye() {
    println!("Goodbye, world!");
}
```

Let's make a very silly change: our own print function. Edit `src/main.rs` to look like this:

```
/// A "Hello, world!" program.
///
/// This is the best implementation of this program to ever exist.

fn main() {
    print("Hello, world!");
    print("Goodbye, world!");
}

fn print(m: &str) {
    println!("{m}")
}
```

Excellent:

```
$ jj log --limit 3
@  vvmrvwuz steve@steveklabnik.com 2024-03-01 17:29:12.000 -06:00
5f858c15
│  refactor printing
│ ◉  povouosx steve@steveklabnik.com 2024-03-01 17:27:14.000 -06:00
28010506
├─╯  remove goodbye message
◉  yykpmnuq steve@steveklabnik.com 2024-03-01 17:07:36.000 -06:00
2b93da0c
│  (empty) add better documentation
```

Everything looks to be in order. Let's rebase our goodbye message change onto our refactor printing change:

```
$ jj rebase -r povouosx -d @
New conflicts appeared in these commits:
  povouosx 793ce8e0 (conflict) remove goodbye message
To resolve the conflicts, start by updating to it:
  jj new povouosxlror
Then use `jj resolve`, or edit the conflict markers in the file
directly.
Once the conflicts are resolved, you may want inspect the result
with `jj diff`.
Then run `jj squash` to move the resolution into the conflicted
commit.
```

Wait a minute, I thought I told you that rebases always succeed. Well... it did:

```
> jj log --limit 3
◉  povouosx steve@steveklabnik.com 2024-03-01 17:30:32.000 -06:00
793ce8e0 conflict
│  remove goodbye message
@  vvmrvwuz steve@steveklabnik.com 2024-03-01 17:29:12.000 -06:00
5f858c15
│  refactor printing
◉  yykpmnuq steve@steveklabnik.com 2024-03-01 17:07:36.000 -06:00
2b93da0c
│  (empty) add better documentation
```

Remember, `@` stays where it is, and we moved a commit ahead of us, so we're good. Go ahead, check out `src/main.rs`, you'll see that it's still just like it was before, with our printer refactoring.

However, you'll notice that in our log output, it says that `povouosx` is now *conflicted*. This is why rebases always succeed in `jj`: if there's a conflict, it doesn't make you stop and fix it, it records that there's a conflict and still performs the rest of the rebase. This is *very* powerful. Even in this case with one change, it lets us handle the conflict when we're ready. We can keep making changes to our current change if we want to:

```
/// A "Hello, world!" program.
///
/// This is the best implementation of this program to ever exist.

fn main() {
    print("Hello, world!");
    print("Goodbye, world!");
}

// a function that prints a message
fn print(m: &str) {
    println!("{m}")
}
```

And then we check our log again:

```
$ jj log --limit 3
Rebased 1 descendant commits onto updated working copy
◉  povouosx steve@steveklabnik.com 2024-03-01 17:49:07.000 -06:00
a912c809 conflict
│  remove goodbye message
@  vvmrvwuz steve@steveklabnik.com 2024-03-01 17:49:07.000 -06:00
d41c079b
│  refactor printing
◉  yykpmnuq steve@steveklabnik.com 2024-03-01 17:07:36.000 -06:00
2b93da0c
│  (empty) add better documentation
```

`jj` automatically rebased `povouosx` again. It's still in conflict. But that's totally okay. We only need to handle it when we're ready. This automatic rebasing behavior only works because `jj` is okay with commits being in conflict. And if we had even more children commits, they'd *all* be rebased, automatically.

Resolving the conflict

The output we got back when the conflict was created gave us some advice:

```
To resolve the conflicts, start by updating to it:
  jj new povouosxlror
Then use `jj resolve`, or edit the conflict markers in the file
directly.
Once the conflicts are resolved, you may want inspect the result
with `jj diff`.
Then run `jj squash` to move the resolution into the conflicted
commit.
```

This advice is good, but also more complex than we need to do right now. Doing this is a great way to handle a complex resolution, where you want to double check what you've done before you apply the changes. But we are just using a small example to make a point. Therefore, we can just edit `povouosx` and remove the conflict markers directly:

```
> jj edit povouosx
Working copy now at: povouosx a912c809 (conflict) remove goodbye
message
Parent commit      : vvmrvwuz d41c079b refactor printing
Added 0 files, modified 1 files, removed 0 files
```

Here's `src/main.rs`:

```
/// A "Hello, world!" program.
///
/// This is the best implementation of this program to ever exist.

fn main() {
<<<<<<<
+++++++
    print("Hello, world!");
    print("Goodbye, world!");
%%%%%%%
     print_hello();
-    print_goodbye();
>>>>>>>
}

// a function that prints a message
fn print(m: &str) {
    println!("{m}")
}
```

`git` uses a combination of `<<<<`, `=====`, and `>>>>` to mark conflicts. `jj` has more rich conflict markers. It still uses the `>>>` and `<<<` s to indicate the start and end, but has two other markers: `+++++++` and `%%%%%%%`. The `+` s indicate the start of a snapshot, and `%` s mark the start of a diff. So we can see that we have two `print` lines, but our changes wanted to remove one of them, but since that's not the same thing, conflict. To resolve this, we apply our own take on the diff to the snapshot:

```
/// A "Hello, world!" program.
///
/// This is the best implementation of this program to ever exist.

fn main() {
    print("Hello, world!");
}

// a function that prints a message
fn print(m: &str) {
    println!("{m}")
}
```

Let's take a look:

```
$ jj st
Working copy changes:
M src\main.rs
Working copy : povouosx 7647f7a0 remove goodbye message
Parent commit: vvmrvwuz d41c079b refactor printing
$ jj log --limit 3
@  povouosx steve@steveklabnik.com 2024-03-01 18:08:23.000 -06:00
7647f7a0
│  remove goodbye message
◉  vvmrvwuz steve@steveklabnik.com 2024-03-01 17:49:07.000 -06:00
d41c079b
│  refactor printing
◉  yykpmnuq steve@steveklabnik.com 2024-03-01 17:07:36.000 -06:00
2b93da0c
│  (empty) add better documentation
```

Conflict resolved!

Automatic rebasing conflict resolution

A wild thing about this though is the combination of conflicted changes and automatic rebasing. Here, I'll show you. First we need to undo our resolution, and then we'll make a new change on top of our conflicted change:

```
> jj undo
New conflicts appeared in these commits:
  povouosx a912c809 (conflict) remove goodbye message
To resolve the conflicts, start by updating to it:
  jj new povouosxlror
Then use `jj resolve`, or edit the conflict markers in the file
directly.
Once the conflicts are resolved, you may want inspect the result
with `jj diff`.
Then run `jj squash` to move the resolution into the conflicted
commit.
Working copy now at: povouosx a912c809 (conflict) remove goodbye
message
Parent commit      : vvmrvwuz d41c079b refactor printing
Added 0 files, modified 1 files, removed 0 files
> jj new povouosxlror --no-edit
Created new commit mlzwmxzs 07bb727d (conflict) (empty) (no
description set)
> jj log --limit 4
◉  mlzwmxzs steve@steveklabnik.com 2024-03-01 18:10:08.000 -06:00
07bb727d conflict
│  (empty) (no description set)
@  povouosx steve@steveklabnik.com 2024-03-01 17:49:07.000 -06:00
a912c809 conflict
│  remove goodbye message
◉  vvmrvwuz steve@steveklabnik.com 2024-03-01 17:49:07.000 -06:00
d41c079b
│  refactor printing
◉  yykpmnuq steve@steveklabnik.com 2024-03-01 17:07:36.000 -06:00
2b93da0c
│  (empty) add better documentation
```

We have our conflict, and then our new change, `mlzwmxzs`, is also in conflict. So fix the conflict in `main.rs` again, and then let's see what happens:

```
> jj log --limit 4
Rebased 1 descendant commits onto updated working copy
◉  mlzwmxzs steve@steveklabnik.com 2024-03-01 18:12:43.000 -06:00
9a4ad229
│  (empty) (no description set)
@  povouosx steve@steveklabnik.com 2024-03-01 18:12:43.000 -06:00
f68d1623
│  remove goodbye message
◉  vvmrvwuz steve@steveklabnik.com 2024-03-01 17:49:07.000 -06:00
d41c079b
│  refactor printing
◉  yykpmnuq steve@steveklabnik.com 2024-03-01 17:07:36.000 -06:00
2b93da0c
│  (empty) add better documentation
```

Not only did we fix our issue, but after we did, `jj` automatically rebased `mlzwmxzs`, and the fix propagated correctly. `mlzwmxzs` is no longer in conflict.

By the way, let's abandon that change, as we don't intend to use it for anything right now:

```
$ jj abandon mlzwmxzs
Abandoned commit mlzwmxzs 9a4ad229 (empty) (no description set)
```

Great, we've cleaned that up.

These behaviors, namely recording conflicts and automatic rebasing, form the behaviors necessary for a very cool `jj` workflow, and that's stacking pull requests. Before we talk about that though, we have to talk about how to use `jj` with GitHub in the first place! Let's go over that next.

Sharing your code with others

One of the best parts about using a version control system is the ability to share that code with other people. So far, we've been using `jj` entirely on our own machine, but it's time to explore how we can interface with tools that let us collaborate.

Here's what we're going to learn:

- Using named branches in jj
- Working with remotes, aka, GitHub
- Adding commits to a pull request
- How to use `jj` with Gerrit rather than GitHub

Using named branches in `jj`

Named branches (or, starting with `jj 0.22`, "bookmarks") are mostly an interoperability feature in `jj`; other than some sort of "main branch" that indicates where shared history lives, other branches aren't necessary to get work done. However, if you use a tool like GitHub, which bases a lot of its functionality around `git` branches, then you'll end up using more than one named branch.

To create a named branch (bookmark) in `jj`, we can use `jj bookmark create`:

```
$ jj bookmark create trunk
$ jj log --limit 2
@  povouosx steve@steveklabnik.com 2024-03-01 18:12:43.000 -06:00
trunk f68d1623
│  remove goodbye message
◉  vvmrvwuz steve@steveklabnik.com 2024-03-01 17:49:07.000 -06:00
d41c079b
│  refactor printing
```

I like the name `trunk` here, but you can use `main` if you prefer, whatever you like really. But if we look on the right hand side of the first log line above, we can see `trunk` as an identifier here. We can use the name `trunk` as a revision or use it in a revset if we'd like:

```
> jj log -r 'ancestors(trunk, 2)'
@  povouosx steve@steveklabnik.com 2024-03-01 18:12:43.000 -06:00
trunk f68d1623
│  remove goodbye message
◉  vvmrvwuz steve@steveklabnik.com 2024-03-01 17:49:07.000 -06:00
d41c079b
│  refactor printing
~
```

One interesting thing about branches in `jj` that's different than branches in `git` is that branches do not automatically move. For example, let's make a new change:

```
> jj new
Working copy now at: moxnkoxx 3f14c03f (empty) (no description set)
Parent commit      : ytkvxlpy 7ec11c41 trunk | remove goodbye
message
> jj log
@  moxnkoxx steve@steveklabnik.com 2024-03-17 17:10:57.000 -05:00
3f14c03f
│  (empty) (no description set)
│ ◉  qtlkpytx steve@steveklabnik.com 2024-03-17 17:09:25.000 -05:00
e6667f9e
├─╯  (empty) (no description set)
◉  ytkvxlpy steve@steveklabnik.com 2024-03-17 17:09:25.000 -05:00
trunk 7ec11c41
│  remove goodbye message
```

Oh look, we have an extra empty commit lying around. That happens sometimes, let's forget about it:

```
> jj abandon qt
Abandoned commit qtlkpytx e6667f9e (empty) (no description set)
> jj log --limit 3
@  moxnkoxx steve@steveklabnik.com 2024-03-17 17:10:57.000 -05:00
3f14c03f
│  (empty) (no description set)
◉  ytkvxlpy steve@steveklabnik.com 2024-03-17 17:09:25.000 -05:00
trunk 7ec11c41
│  remove goodbye message
◉  krmulszn steve@steveklabnik.com 2024-03-17 17:06:59.000 -05:00
e98c1626
│  refactor printing
```

Even though `@` has moved to `moxnkoxx`, `trunk` is still at `ytkvxlpy`. This behavior is a bit surprising for folks coming from `git`, though it fits in with `jj` more nicely, I think.

Regardless, let's update `trunk` to point at `@`:

```
$ jj bookmark set trunk
Moved 1 bookmarks to pzkrzopz fcf669c5 trunk | (empty) (no
description set)
$ jj log --limit 2
@  pzkrzopz steve@steveklabnik.com 2024-03-01 22:41:37.000 -06:00
trunk fcf669c5
│  (empty) (no description set)
◉  povouosx steve@steveklabnik.com 2024-03-01 18:12:43.000 -06:00
f68d1623
│  remove goodbye message
```

If you want to replicate `git`'s behavior, typing an additional command after each change is done feels like overkill. But I would argue this is not the right way to use `jj`; as you'll see, we'll be either re-writing commits at the tip of branches, or doing multiple steps of work before updating where a branch points. In practice, it means I check the branch status *before* pushing code, rather than as I work. That is, the branch name tends to sit at the same change as the remote server, and when it's time to update the remote, that's when I update things locally.

Speaking of remotes, let's talk about that next.

Working with remotes, e.g., GitHub

We're going to talk about working with remote `git` servers, and using GitHub as an example. The same principles apply to any given `git` server, though.

Pushing our code to GitHub

The first thing to do is to create the remote server. I have made a GitHub project at https://github.com/steveklabnik/jj-hello-world. I'm adding it as an upstream like this:

```
> jj git remote add origin git@github.com:steveklabnik/jj-hello-
world.git
```

Before we push our commit up, we need to fix our repository:

```
$ jj log --limit 2
@  pzkrzopz steve@steveklabnik.com 2024-03-01 22:41:37.000 -06:00
trunk fcf669c5
│  (empty) (no description set)
◉  povouosx steve@steveklabnik.com 2024-03-01 18:12:43.000 -06:00
f68d1623
│  remove goodbye message
```

Let's swap `@` back to the previous change, and then abandon this one. We can do that like this:

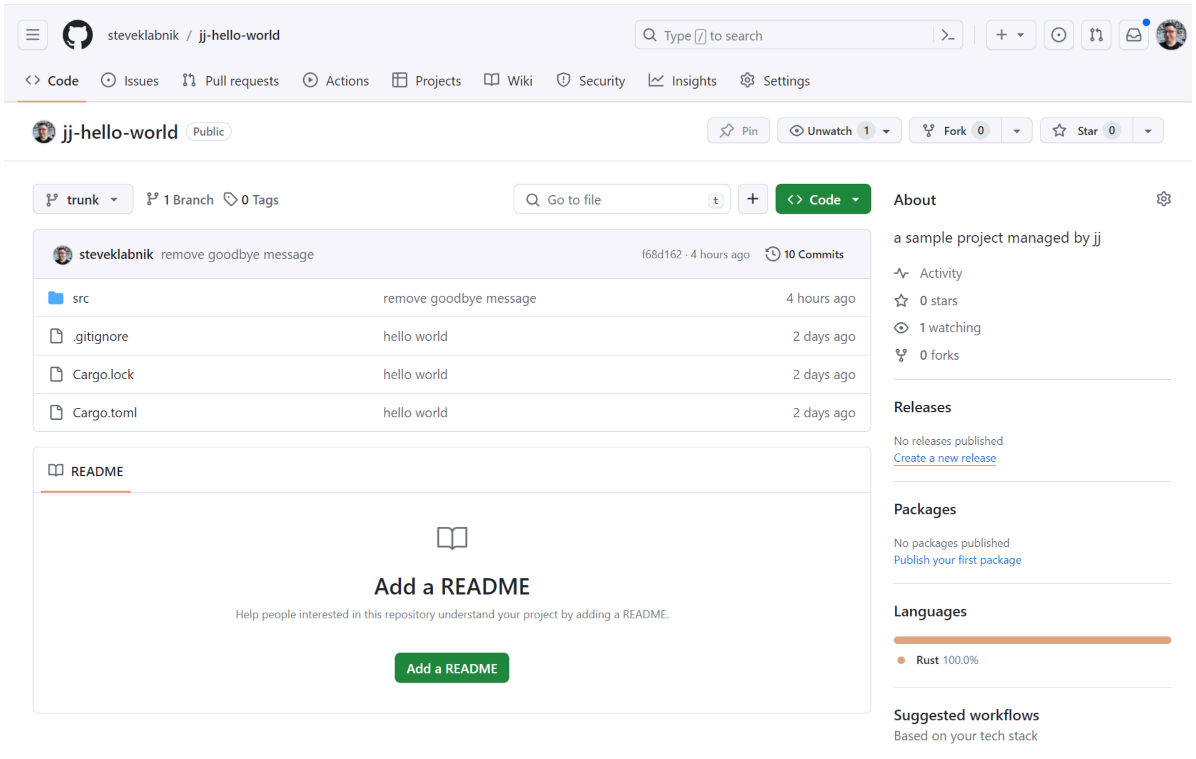
```
$ jj edit @-
Working copy now at: povouosx f68d1623 remove goodbye message
Parent commit      : vvmrvwuz d41c079b refactor printing
$ jj bookmark set trunk --allow-backwards
Moved 1 bookmarks to povouosx f68d1623 | remove goodbye message
$ jj abandon pzkrzopz
Abandoned commit pzkrzopz fcf669c5 (empty) (no description set)
$ jj log --limit 2
@  povouosx steve@steveklabnik.com 2024-03-01 18:12:43.000 -06:00
trunk f68d1623
│  remove goodbye message
◉  vvmrvwuz steve@steveklabnik.com 2024-03-01 17:49:07.000 -06:00
d41c079b
│  refactor printing
```

We need the `--allow-backwards` flag to set the `trunk` branch to the previous commit because it is a dangerous operation: if we had pushed `trunk`, things would get weird when we try and push now. We've kept it all local, so there's no issues with doing this.

Anyway, let's push `trunk` up to GitHub:

```
$ jj git push
Changes to push to origin:
  Add bookmark trunk to f68d16233bdc
Warning: The working-copy commit in workspace 'default' became
immutable, so a new commit has been created on top of it.
Working copy now at: znurnwmk f853107d (empty) (no description set)
Parent commit      : povouosx f68d1623 | remove goodbye message
```

And now our project is up on GitHub!



steveklabnik remove goodbye message	f68d162 · 4 hours ago	10 Commits
src	remove goodbye message	4 hours ago
.gitignore	hello world	2 days ago
Cargo.lock	hello world	2 days ago
Cargo.toml	hello world	2 days ago

Add a README

Help people interested in this repository understand your project by adding a README.

About

a sample project managed by jj

Activity
0 stars
1 watching
0 forks

Releases

No releases published

Packages

No packages published

Languages

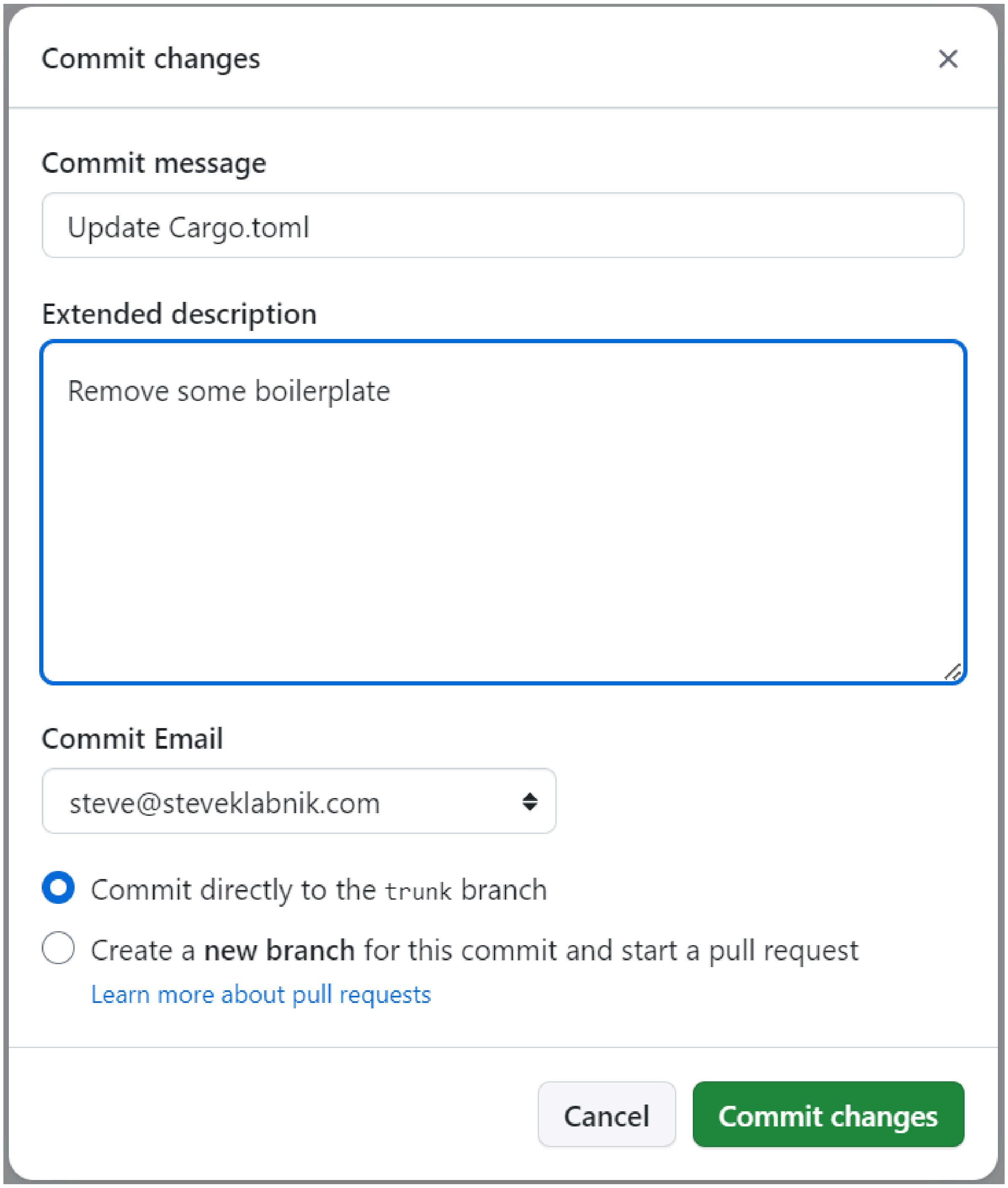
Rust 100.0%

Suggested workflows

Based on your tech stack

Updating the `trunk` branch from GitHub

If you collaborate on a project, as commits land on the main branch, you'll want to update your local copy of that branch. I'm going to make a change in the GitHub UI:


Commit changes

Commit message

Update Cargo.toml

Extended description

Remove some boilerplate

Commit Email

steve@steveklabnik.com

[✓ Commit directly to the `trunk` branch]

Create a **new branch** for this commit and start a pull request

All I did was update the `Cargo.toml` to remove some comments. If you're following along, you can make any change you'd like.

Let's fetch those changes:

```
$ jj git fetch
bookmark: trunk@origin [updated] tracked
$ jj log --limit 3
◉  ksrmwuon steve@steveklabnik.com 2024-03-01 23:10:35.000 -06:00
trunk e202b67c
│  Update Cargo.toml
│ @  znurnwmk steve@steveklabnik.com 2024-03-01 18:15:00.000
f853107d
├─╯  (empty) (no description set)
@  povouosx steve@steveklabnik.com 2024-03-01 18:12:43.000 -06:00
f68d1623
│  remove goodbye message
~
```

In this instance, `trunk` did move to the new commit: we asked `jj` to fetch information from our origin, and so it's adjusted things to match. However, `@` is still at our current commit (ie. the empty commit created by `jj git push`).

Let's fix that:

```
$ jj new trunk
Working copy now at: vmunwxsk be917d2e (empty) (no description set)
Parent commit      : ksrmwuon e202b67c trunk | Update Cargo.toml
Added 0 files, modified 1 files, removed 0 files
```

Now we're working ahead of our `trunk`.

Creating a pull request

On GitHub, pull requests are tied to a branch. But if you're doing this `jj`-native workflow, you aren't really thinking about branch names. Does this advantage go away when you start working with pull requests? Not particularly.

Let's make this empty change we're on into a real change. Update `src/main.rs` with a new comment:

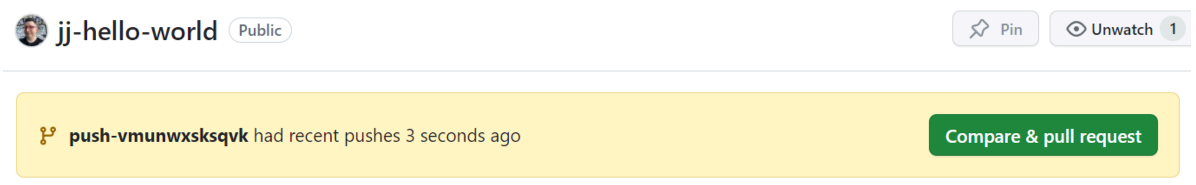
```
/// A "Hello, world!" program.
///
/// This is the best implementation of this program to ever exist.

/// add documentation for main
fn main() {
    print("Hello, world!");
}

// a function that prints a message
fn print(m: &str) {
    println!("{m}")
}
```

Then let's add a description, and push our change to GitHub so we can make a PR:

```
$ jj describe -m "add a comment to main"
Working copy now at: vmunwxsk 9410db49 add a comment to main
Parent commit      : ksrmwuon e202b67c trunk | Update Cargo.toml
$ jj git push -c @
Creating bookmark push-vmunwxsksqvk for revision vmunwxsksqvk
Changes to push to origin:
  Add bookmark push-vmunwxsksqvk to 9410db49f9ba
$ jj log
@  vmunwxsk steve@steveklabnik.com 2024-03-02 08:27:30.000 -06:00
push-vmunwxsksqvk 9410db49
│  add a comment to main
◉  ksrmwuon steve@steveklabnik.com 2024-03-01 23:10:35.000 -06:00
trunk e202b67c
│  Update Cargo.toml
~
```

We've used `jj git push` to push code to a `git` remote before, but the `-c` flag is new: we're asking it to create us a new branch, from the revision `@`. And so it did, and gave us the name `push-vmunwxsksqvk`. This is our change ID; it's longer because change IDs are actually longer than what's been displayed to us, there's just never been a reason to show the whole ID, since any unique prefix works: `vmunwxsk` is just as much a unique prefix of `vmunwxsksqvk` as `v` is, it's just not the shortest unique prefix.

We can now make a pull request out of this:

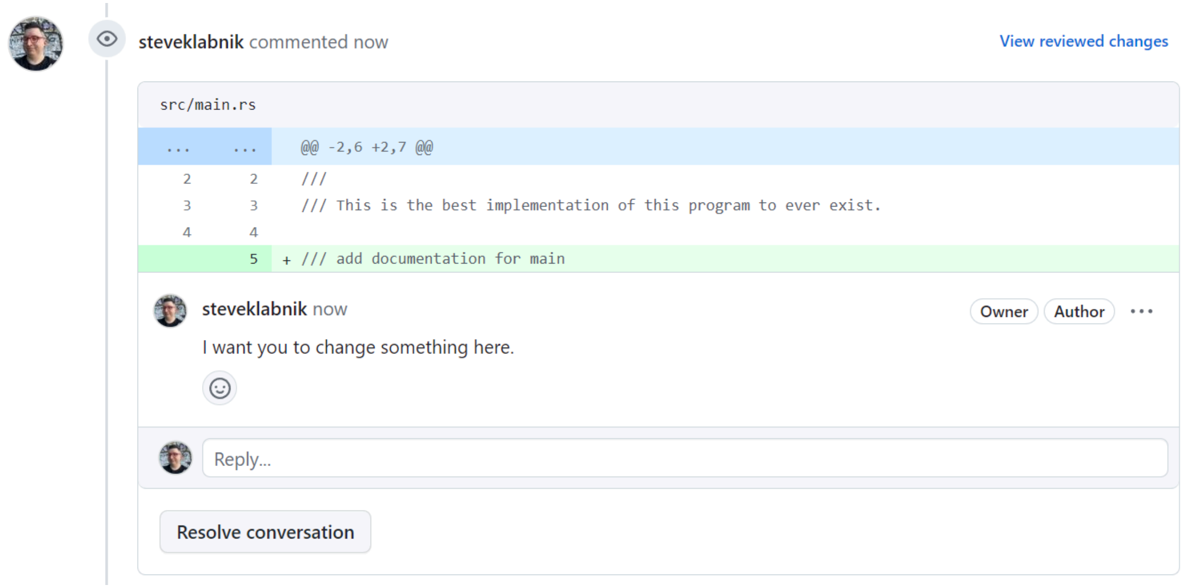


push-vmunwxsksqvk had recent pushes 3 seconds ago

If you'd like to view this PR, you can find it here, though by the time you look at it, some changes will have been made! Even the smallest pull requests get feedback sometimes, and we're gonna learn two ways of dealing with review comments in the next section.

Responding to pull request feedback

Oh no! Someone has asked for changes to our pull request.


steveklabnik commented now

```
src/main.rs
@@ -2,6 +2,7 @@
///
/// This is the best implementation of this program to ever exist.

+ /// add documentation for main
```

steveklabnik now

I want you to change something here.

We have two ways of making our change, and it depends on what the project's maintainers prefer. Due to the way that GitHub shows or minimizes comments on a pull request, some projects prefer that you never modify commits that you've pushed to a pull request, and only want you to add commits to fix problems. Other projects are okay with you modifying history, or even actively want one commit per pull request. We can do both, but the way we do it looks a little bit different.

Adding commits to a PR

Adding commits to a PR is easy, but works *just* differently enough in `jj` that it can be confusing at first. Let me explain.

Let's create a new change:

```
$ jj new -m "respond to feedback"
Working copy now at: nzsvmmzl 3b663200 (empty) respond to feedback
Parent commit      : vmunwxsk 9410db49 push-vmunwxsksqvk | add a
comment to main
```

And change the text of the comment in `src/main.rs`:

```
/// A "Hello, world!" program.
///
/// This is the best implementation of this program to ever exist.

/// The main function runs when our program starts
fn main() {
    print("Hello, world!");
}

// a function that prints a message
fn print(m: &str) {
    println!("{m}")
}
```

Our change is ready, but one thing is missing:

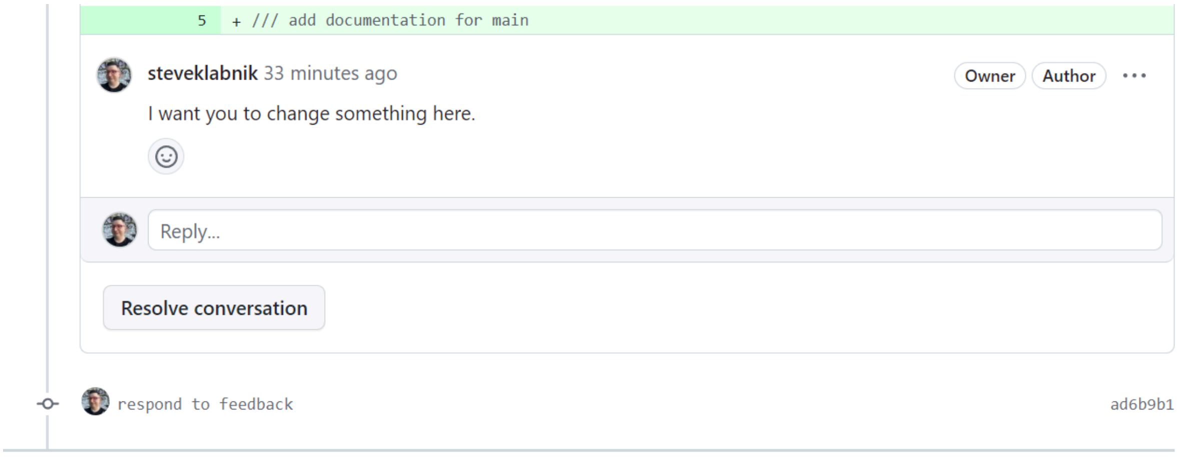
```
$ jj log
@  nzsvmmzl steve@steveklabnik.com 2024-03-02 09:22:40.000 -06:00
ad6b9b14
│  respond to feedback
◉  vmunwxsk steve@steveklabnik.com 2024-03-02 08:27:30.000 -06:00
push-vmunwxsksqvk 9410db49
│  add a comment to main
◉  ksrmwuon steve@steveklabnik.com 2024-03-01 23:10:35.000 -06:00
trunk e202b67c
│  Update Cargo.toml
~
```

Remember, `jj new` won't move any branches, and so if we push, nothing happens:

```
$ jj git push
Warning: No bookmarks found in the default push revset:
remote_bookmarks(remote=origin)..@
Nothing changed.
```

First we have to update the branch to point at our new commit, and then push:

```
$ jj bookmark set push-vmunwxsksqvk
Moved 1 bookmarks to nzsvmmzl ad6b9b14 push-vmunwxsksqvk* | respond
to feedback
$ jj git push
Changes to push to origin:
  Move forward bookmark push-vmunwxsksqvk from 9410db49f9ba to
ad6b9b149f88
```



```
5  + /// add documentation for main
```

steveklabnik 33 minutes ago

I want you to change something here.

respond to feedback ad6b9b1

Since we added a commit, the review comment is still there, which is why people like this workflow.

Now, when people learn about this behavior for the first time, they're often a little annoyed. It sounds like you have to do an extra step each time you change your pull request. But I've found that in practice, this overhead affects small PRs more than larger ones. The real mental change is to make sure that your branch is pointing where you want it to just before you push, not after each commit. In other words, the workflow is not:

1. Make change
2. Update branch
3. Make change
4. Update branch
5. Push

It is:

1. Make change

2. Make change
3. Update branch
4. Push

Since I am always looking things over before I push them up, I'm already going to notice if my branch is out of date, and if you forget, the worst case is that nothing gets pushed, which is a reminder to go and update the branch anyway.

The next workflow eliminates this extra "update the branch" step, but comes with its own challenges. Let's look at rebasing pull requests instead of adding new commits to them.

Rebasing a PR

First, we have to undo what we just did. Here's where we are:

```
> jj log
@  nzsvmmzl steve@steveklabnik.com 2024-03-02 09:22:40.000 -06:00
push-vmunwxsksqvk ad6b9b14
│  respond to feedback
◉  vmunwxsk steve@steveklabnik.com 2024-03-02 08:27:30.000 -06:00
9410db49
│  add a comment to main
◉  ksrmwuon steve@steveklabnik.com 2024-03-01 23:10:35.000 -06:00
trunk e202b67c
│  Update Cargo.toml
~
```

So let's move the branch backwards, then abandon our new change:

```
$ jj bookmark set push-vmunwxsksqvk -r @- --allow-backwards
Moved 1 bookmarks to vmunwxsk 9410db49 push-vmunwxsksqvk* | add a
comment to main
$ jj edit vmunwxsk
Working copy now at: vmunwxsk 9410db49 push-vmunwxsksqvk* | add a
comment to main
Parent commit      : ksrmwuon e202b67c trunk | Update Cargo.toml
Added 0 files, modified 1 files, removed 0 files
$ jj abandon nzsvmmzl
Abandoned commit nzsvmmzl ad6b9b14 push-vmunwxsksqvk@origin |
respond to feedback
$ jj log
@  vmunwxsk steve@steveklabnik.com 2024-03-02 08:27:30.000 -06:00
push-vmunwxsksqvk* 9410db49
│  add a comment to main
◉  ksrmwuon steve@steveklabnik.com 2024-03-01 23:10:35.000 -06:00
trunk e202b67c
│  Update Cargo.toml
~
```

Nice. We can see that the branch name has an asterisk by it now; because we are tracking this branch to a remote, `jj` is noting that our understanding of this branch and the remote's understanding are different.

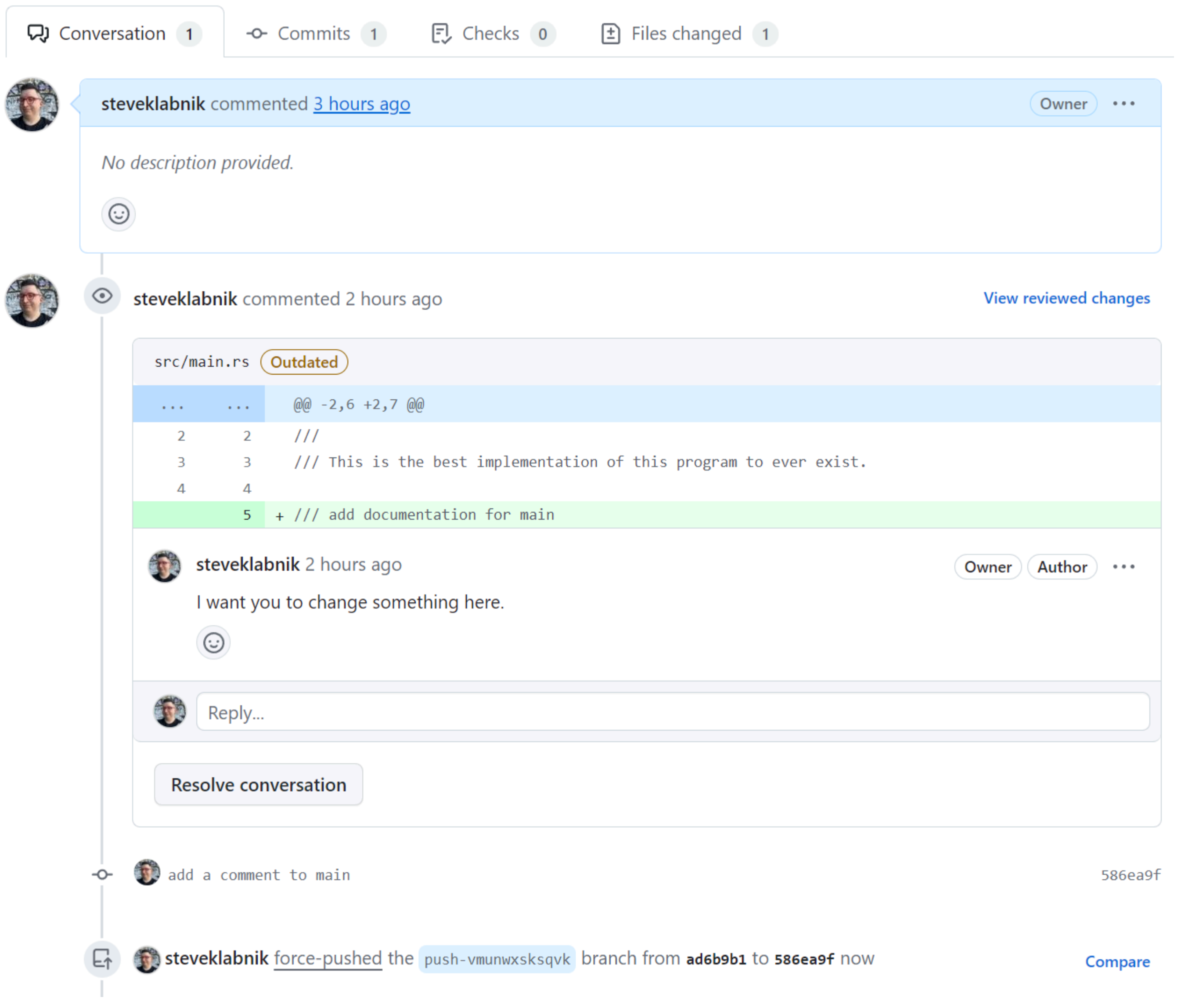
Let's make it even more different: let's edit `src/main.rs` again:

```
/// A "Hello, world!" program.
///
/// This is the best implementation of this program to ever exist.

/// The main function runs when our program starts
fn main() {
    print("Hello, world!");
}

// a function that prints a message
fn print(m: &str) {
    println!("{m}")
}
```

Same as before. Since `jj` is tracking changes that we've made, our commit has already been "rebased" in a sense. So we can just push:

```
> jj git push
Changes to push to origin:
  Move sideways bookmark push-vmunwxsksqvk from ad6b9b149f88 to
586ea9fd213f
```

We can see that reflected on GitHub:



Conversation 1 | Commits 1 | Checks 0 | Files changed 1

steveklabnik commented 3 hours ago — Owner

No description provided.

steveklabnik commented 2 hours ago — View reviewed changes

src/main.rs — Outdated

```
@@ -2,6 +2,7 @@
 ///
 /// This is the best implementation of this program to ever exist.

+/// add documentation for main
```

steveklabnik 2 hours ago — Owner — Author

I want you to change something here.

Reply...

Resolve conversation

add a comment to main — 586ea9f

steveklabnik force-pushed the `push-vmunwxsksqvk` branch from **ad6b9b1** to **586ea9f** now — Compare

Only one commit, and it shows that our comment is on an outdated diff.

This is really "rewriting" more than "rebasing," but rebasing also involve rewriting history and you may additionally want to rebase

Rebasing with multiple changes

Sometimes you have pull requests with more than one commit, though. What if we had more than one change we wanted to make? Let's undo our change. Because this is so small, I'm going to do it by hand, changing `src/main.rs`:

```
/// A "Hello, world!" program.
///
/// This is the best implementation of this program to ever exist.

/// add documentation for main
fn main() {
    print("Hello, world!");
}

// a function that prints a message
fn print(m: &str) {
    println!("{m}")
}
```

Let's make a new commit for some other sort of change. First we need to `jj new`:

```
$  jj new -m "add a new function"
Working copy now at: msmntwvo baaa23e8 (empty) add a new function
Parent commit      : vmunwxsk 6da57c93 push-vmunwxsksqvk* | add a
comment to main
```

And then add some new functionality:

```
/// A "Hello, world!" program.
///
/// This is the best implementation of this program to ever exist.

/// add documentation for main
fn main() {
    print("Hello, world!");
    print("Goodbye, world!");
}

// a function that prints a message
fn print(m: &str) {
    println!("{m}")
}
```

Let's update our branch and push:

```
> jj bookmark set push-vmunwxsksqvk
Moved 1 bookmarks to msmntwvo 8f7dcd91 push-vmunwxsksqvk* | add a
new function
> jj git push
Changes to push to origin:
  Move sideways bookmark push-vmunwxsksqvk from 586ea9fd213f to
8f7dcd91ecbf
```

We now have two changes again. So what happens when we address our review? Well, since we're okay with rebasing, we can just edit that commit directly:

```
$ jj edit vmunwxsk
Working copy now at: vmunwxsk 6da57c93 add a comment to main
Parent commit      : ksrmwuon e202b67c trunk | Update Cargo.toml
Added 0 files, modified 1 files, removed 0 files
```

and update `src/main.rs`:

```
/// A "Hello, world!" program.
///
/// This is the best implementation of this program to ever exist.

/// The main function runs when our program starts
fn main() {
    print("Hello, world!");
    print("Goodbye, world!");
}

// a function that prints a message
fn print(m: &str) {
    println!("{m}")
}
```

And check our work:

```
$ jj st
 jj st
Rebased 1 descendant commits onto updated working copy
Working copy changes:
M src\main.rs
Working copy : vmunwxsk f6f7dce9 add a comment to main
Parent commit: ksrmwuon e202b67c trunk | Update Cargo.toml
```

There's that automatic rebase again! We don't need to do anything with our "add a new function" change, as it isn't conflicted. So we can just go ahead and push:

```
$ jj log
◉  msmntwvo steve@steveklabnik.com 2024-03-02 11:47:08.000 -06:00
push-vmunwxsksqvk* 752534be
│  add a new function
@  vmunwxsk steve@steveklabnik.com 2024-03-02 11:47:08.000 -06:00
f6f7dce9
│  add a comment to main
◉  ksrmwuon steve@steveklabnik.com 2024-03-01 23:10:35.000 -06:00
trunk e202b67c
│  Update Cargo.toml
~
$ jj next --edit
Working copy now at: msmntwvo 752534be push-vmunwxsksqvk* | add a
new function
Parent commit      : vmunwxsk f6f7dce9 add a comment to main
Added 0 files, modified 1 files, removed 0 files
$ jj git push
Changes to push to origin:
  Move sideways bookmark push-vmunwxsksqvk from 8f7dcd91ecbf to
752534beb39f
```

And now we're good! Just that easy. If we didn't want to move `@`, we could have done `jj git push -b push-vmunwxsksqvk` to push that specific branch, but I like not staying in the middle of a branch once I'm done with that work.

Recap

We learned two different ways of handling pull request feedback: adding more commits, and rewriting commits. While both of these ways work, they both have different drawbacks. In the next chapter, we're going to explore some more advanced ways of working that allow us to mitigate some of the drawbacks, as well as talk about working with other tools that let us work with `jj` in a nicer way.

More advanced workflows

One of the best parts about using a version control system is the ability to share that code with other people. So far, we've been using jj entirely on our own machine, but it's time to explore how we can interface with tools that let us collaborate.

Here's what we're going to learn:

- Working on all your branches simultaneously
- A pull request workflow called "Stacked PRs"
- Workspaces that let you have multiple local checkouts
- Using `git` and `jj` at the same time with colocated repositories

Working on all of your branches simultaneously

I think this is one of the first things I saw about `jj` that made me go "wait, WHAT?!?" I was asking questions on the `jj` Discord, and one of the developers, Austin, mentioned that

> Also. I can rebase all my branches simultaneously too.

This section is going to explain what he meant by that, and how he uses `jj` to manage multiple pull requests at the same time.

First, we'll do some set up to have a few pull requests going on at the same time. Then we'll show you how you can develop against all of them simultaneously, and when your upstream updates, you can rebase them all simultaneously.

The set up

Let's check out our example project to make sure we're on the same place:

```
> jj log
@  msmntwvo steve@steveklabnik.com 2024-03-02 11:47:08.000 -06:00
push-vmunwxsksqvk 752534be
│  add a new function
◉  vmunwxsk steve@steveklabnik.com 2024-03-02 11:47:08.000 -06:00
f6f7dce9
│  add a comment to main
◉  ksrmwuon steve@steveklabnik.com 2024-03-01 23:10:35.000 -06:00
trunk e202b67c
│  Update Cargo.toml
```

Right! We have an outstanding pull request from a previous section. If you have made some of your own changes, maybe do this next section an extra time, so that you have enough outstanding branches. Austin's original example had five, so that's how many I'm going to make. If you want to make fewer, two works too, or you can go a little harder and make even more branches. Up to you!

Let's make some more branches! We want to start them all from `trunk`, not on each other:

```
> jj new trunk
Working copy now at: opwqpunl 7ede4eb9 (empty) (no description set)
Parent commit      : ksrmwuon e202b67c trunk | Update Cargo.toml
Added 0 files, modified 1 files, removed 0 files
~/Documents/GitHub/sample-jj-project/hello-world> jj log
@  opwqpunl steve@steveklabnik.com 2024-03-17 14:12:52.000 -05:00
7ede4eb9
│  (empty) (no description set)
│ ◉  msmntwvo steve@steveklabnik.com 2024-03-02 11:47:08.000 -06:00
push-vmunwxsksqvk 752534be
│ │  add a new function
│ ◉  vmunwxsk steve@steveklabnik.com 2024-03-02 11:47:08.000 -06:00
f6f7dce9
├─╯  add a comment to main
◉  ksrmwuon steve@steveklabnik.com 2024-03-01 23:10:35.000 -06:00
trunk e202b67c
│  Update Cargo.toml
~
```

We've now started a new branch from `trunk`. Let's give it a description, add a second commit with its own description, and then make a pull request out of it. Feel free to make changes in the code in here if you'd like, it doesn't matter for our purposes.

```
> jj describe -m "display the birthday date on the settings page"
Working copy now at: opwqpunl 1a66beb1 (empty) display the birthday
date on the settings page
Parent commit      : ksrmwuon e202b67c trunk | Update Cargo.toml
> jj new -m "have galactus query eks with time range"
Working copy now at: yxxppztp 3d123151 (empty) have galactus query
eks with time range
Parent commit      : opwqpunl 1a66beb1 (empty) display the birthday
date on the settings page
> jj git push -c @
Creating branch push-yxxppztpoyqq for revision @
Branch changes to push to origin:
  Add branch push-yxxppztpoyqq to 3d1231518dbf
> jj log
@  yxxppztp steve@steveklabnik.com 2024-03-17 14:18:00.000 -05:00
push-yxxppztpoyqq 3d123151
│  (empty) have galactus query eks with time range
◉  opwqpunl steve@steveklabnik.com 2024-03-17 14:15:58.000 -05:00
1a66beb1
│  (empty) display the birthday date on the settings page
│ ◉  msmntwvo steve@steveklabnik.com 2024-03-02 11:47:08.000 -06:00
push-vmunwxsksqvk 752534be
│ │  add a new function
│ ◉  vmunwxsk steve@steveklabnik.com 2024-03-02 11:47:08.000 -06:00
f6f7dce9
├─╯  add a comment to main
◉  ksrmwuon steve@steveklabnik.com 2024-03-01 23:10:35.000 -06:00
trunk e202b67c
│  Update Cargo.toml
~
```

Great! We have two pull requests. Do this again at least once, but maybe two or three times. It's cooler with more branches, trust me.

```
> jj new trunk -m "prepare to deploy to the cloud"
Working copy now at: tmnmvxyy 105cf6b5 (empty) prepare to deploy to
the cloud
Parent commit      : ksrmwuon e202b67c trunk | Update Cargo.toml
> jj new -m "various fixes"
Working copy now at: rxpztwms 902a6cd2 (empty) various fixes
Parent commit      : tmnmvxyy 105cf6b5 (empty) prepare to deploy to
the cloud
> jj git push -c @
Creating branch push-rxpztwmsszvk for revision @
Branch changes to push to origin:
  Add branch push-rxpztwmsszvk to 902a6cd22f30
```

Your `jj log` will look like this:

```
> jj log
@  ymvptyyq steve@steveklabnik.com 2024-03-17 14:25:31.000 -05:00
push-ymvptyyqmyul 728dbb1e
│  (empty) fixing all the breakage from updating dependencies
◉  xulymzyp steve@steveklabnik.com 2024-03-17 14:25:14.000 -05:00
1f7c69a5
│  (empty) updating dependencies
│ ◉  zxyukunn steve@steveklabnik.com 2024-03-17 14:24:56.000 -05:00
push-zxyukunnwolo 30081a6b
│ │  (empty) first 80% done
│ ◉  tzsloruo steve@steveklabnik.com 2024-03-17 14:24:21.000 -05:00
7c02f6ce
├─╯  (empty) another feature
│ ◉  rxpztwms steve@steveklabnik.com 2024-03-17 14:23:00.000 -05:00
push-rxpztwmsszvk 902a6cd2
│ │  (empty) various fixes
│ ◉  tmnmvxyy steve@steveklabnik.com 2024-03-17 14:22:15.000 -05:00
105cf6b5
├─╯  (empty) prepare to deploy to the cloud
│ ◉  yxxppztp steve@steveklabnik.com 2024-03-17 14:18:00.000 -05:00
push-yxxppztpoyqq 3d123151
│ │  (empty) have galactus query eks with time range
│ ◉  opwqpunl steve@steveklabnik.com 2024-03-17 14:15:58.000 -05:00
1a66beb1
├─╯  (empty) display the birthday date on the settings page
│ ◉  msmntwvo steve@steveklabnik.com 2024-03-02 11:47:08.000 -06:00
push-vmunwxsksqvk 752534be
│ │  add a new function
│ ◉  vmunwxsk steve@steveklabnik.com 2024-03-02 11:47:08.000 -06:00
f6f7dce9
├─╯  add a comment to main
◉  ksrmwuon steve@steveklabnik.com 2024-03-01 23:10:35.000 -06:00
trunk e202b67c
│  Update Cargo.toml
```

That's a lot of branches!

We could do what we do with `git`, and just work on each branch individually. But we can do more interesting things than that: let's work on every branch at the same time.

Working on every branch

So, here's how we can work on every branch at the same time: we create a merge with the parents of *every* branch we have. My command looks like this, but you'll have to use the appropriate change IDs for what your repository looks like. I'm also using the short form of each name, which helps when you have five of them! Anyway:

```
> jj new ym z r yx m -m "merge: steve's branch"
Working copy now at: xnutwmso 695806ff (empty) merge: steve's branch
Parent commit      : ymvptyyq 728dbb1e push-ymvptyyqmyul | (empty)
fixing all the breakage from updating dependencies
Parent commit      : zxyukunn 30081a6b push-zxyukunnwolo | (empty)
first 80% done
Parent commit      : rxpztwms 902a6cd2 push-rxpztwmsszvk | (empty)
various fixes
Parent commit      : yxxppztp 3d123151 push-yxxppztpoyqq | (empty)
have galactus query eks with time range
Parent commit      : msmntwvo 752534be push-vmunwxsksqvk | add a new
function
Added 0 files, modified 1 files, removed 0 files
```

Whew! What a change! Five parents. Let's create an extra one so we can use our squash-style workflow more easily: we temporarily work on a `@` change, and then `jj squash` diffs back into whichever parent makes the most sense.

```
> jj new
Working copy now at: nllzosqm 85324040 (empty) (no description set)
Parent commit      : xnutwmso 695806ff (empty) merge: steve's branch
```

Check out this `jj log`:

```
> jj log
@  nllzosqm steve@steveklabnik.com 2024-03-17 14:36:36.000 -05:00
85324040
│  (empty) (no description set)
◉         xnutwmso steve@steveklabnik.com 2024-03-17 14:30:52.000
-05:00 695806ff
├───┬───╮  (empty) merge: steve's branch
│ │ │ │ ◉  msmntwvo steve@steveklabnik.com 2024-03-02 11:47:08.000
-06:00 push-vmunwxsksqvk 752534be
│ │ │ │ │  add a new function
│ │ │ │ ◉  vmunwxsk steve@steveklabnik.com 2024-03-02 11:47:08.000
-06:00 f6f7dce9
│ │ │ │ │  add a comment to main
│ │ │ ◉ │  yxxppztp steve@steveklabnik.com 2024-03-17 14:18:00.000
-05:00 push-yxxppztpoyqq 3d123151
│ │ │ │ │  (empty) have galactus query eks with time range
│ │ │ ◉ │  opwqpunl steve@steveklabnik.com 2024-03-17 14:15:58.000
-05:00 1a66beb1
│ │ │ ├─╯  (empty) display the birthday date on the settings page
│ │ ◉ │  rxpztwms steve@steveklabnik.com 2024-03-17 14:23:00.000
-05:00 push-rxpztwmsszvk 902a6cd2
│ │ │ │  (empty) various fixes
│ │ ◉ │  tmnmvxyy steve@steveklabnik.com 2024-03-17 14:22:15.000
-05:00 105cf6b5
│ │ ├─╯  (empty) prepare to deploy to the cloud
│ ◉ │  zxyukunn steve@steveklabnik.com 2024-03-17 14:24:56.000
-05:00 push-zxyukunnwolo 30081a6b
│ │ │  (empty) first 80% done
│ ◉ │  tzsloruo steve@steveklabnik.com 2024-03-17 14:24:21.000
-05:00 7c02f6ce
│ ├─╯  (empty) another feature
◉ │  ymvptyyq steve@steveklabnik.com 2024-03-17 14:25:31.000 -05:00
push-ymvptyyqmyul 728dbb1e
│ │  (empty) fixing all the breakage from updating dependencies
◉ │  xulymzyp steve@steveklabnik.com 2024-03-17 14:25:14.000 -05:00
1f7c69a5
├─╯  (empty) updating dependencies
◉  ksrmwuon steve@steveklabnik.com 2024-03-01 23:10:35.000 -06:00
trunk e202b67c
│  Update Cargo.toml
```

Glorious!

So now our working directory has all of our changes in it. We can make changes, and then `jj squash` them into the appropriate branch. If we decide we want a new change at the head of any of these branches, we'll need to use a rebase, but it's not too bad:

```
> jj new z -m "second 80% done"
Working copy now at: kvupxvpv 2ea49586 (empty) second 80% done
Parent commit      : zxyukunn 30081a6b push-zxyukunnwolo | (empty)
first 80% done
Added 0 files, modified 1 files, removed 0 files
> jj log
    @  kvupxvpv steve@steveklabnik.com 2024-03-17 14:43:28.000
-05:00 46cb6847
    │ (empty) second 80% done
  ◉ │      xnutwmso steve@steveklabnik.com 2024-03-17 14:30:52.000
-05:00 695806ff
╭─┼─┬─┬─╮  (empty) merge: steve's branch
◉ │ │ │ │  msmntwvo steve@steveklabnik.com 2024-03-02 11:47:08.000
-06:00 push-vmunwxsksqvk 752534be
│ │ │ │ │  add a new function
◉ │ │ │ │  vmunwxsk steve@steveklabnik.com 2024-03-02 11:47:08.000
-06:00 f6f7dce9
│ │ │ │ │  add a comment to main
│ │ │ │ ◉  yxxppztp steve@steveklabnik.com 2024-03-17 14:18:00.000
-05:00 push-yxxppztpoyqq 3d123151
│ │ │ │ │  (empty) have galactus query eks with time range
│ │ │ │ ◉  opwqpunl steve@steveklabnik.com 2024-03-17 14:15:58.000
-05:00 1a66beb1
├───────╯  (empty) display the birthday date on the settings page
│ │ │ ◉  rxpztwms steve@steveklabnik.com 2024-03-17 14:23:00.000
-05:00 push-rxpztwmsszvk 902a6cd2
│ │ │ │  (empty) various fixes
│ │ │ ◉  tmnmvxyy steve@steveklabnik.com 2024-03-17 14:22:15.000
-05:00 105cf6b5
├─────╯  (empty) prepare to deploy to the cloud
│ │ ◉  zxyukunn steve@steveklabnik.com 2024-03-17 14:24:56.000
-05:00 push-zxyukunnwolo 30081a6b
│ │ │  (empty) first 80% done
│ │ ◉  tzsloruo steve@steveklabnik.com 2024-03-17 14:24:21.000
-05:00 7c02f6ce
├───╯  (empty) another feature
│ ◉  ymvptyyq steve@steveklabnik.com 2024-03-17 14:25:31.000 -05:00
push-ymvptyyqmyul 728dbb1e
│ │  (empty) fixing all the breakage from updating dependencies
│ ◉  xulymzyp steve@steveklabnik.com 2024-03-17 14:25:14.000 -05:00
1f7c69a5
├─╯  (empty) updating dependencies
◉  ksrmwuon steve@steveklabnik.com 2024-03-01 23:10:35.000 -06:00
trunk e202b67c
│  Update Cargo.toml
```

Yikes! Don't worry, we can fix that with a rebase:

```
> jj rebase -r xn -d m -d ym -d yx -d r -d kv
```

We want to rebase the revision `xn` with the following "destination" revisions: `m`, `ym`, `yx`, `r`, and `kv`. Since we have multiple parents, that's what will happen:

```
> jj log
◉         xnutwmso steve@steveklabnik.com 2024-03-17 15:16:36.000
-05:00 da67dfe1
├─┬─┬─┬─╮  (empty) merge: steve's branch
│ │ │ │ @  kvupxvpv steve@steveklabnik.com 2024-03-17 15:15:20.000
-05:00 2ea49586
│ │ │ │ │  (empty) second 80% done
│ │ │ │ ◉  zxyukunn steve@steveklabnik.com 2024-03-17 14:24:56.000
-05:00 push-zxyukunnwolo 30081a6b
│ │ │ │ │  (empty) first 80% done
│ │ │ │ ◉  tzsloruo steve@steveklabnik.com 2024-03-17 14:24:21.000
-05:00 7c02f6ce
│ │ │ │ │  (empty) another feature
│ │ │ ◉ │  rxpztwms steve@steveklabnik.com 2024-03-17 14:23:00.000
-05:00 push-rxpztwmsszvk 902a6cd2
│ │ │ │ │  (empty) various fixes
│ │ │ ◉ │  tmnmvxyy steve@steveklabnik.com 2024-03-17 14:22:15.000
-05:00 105cf6b5
│ │ │ ├─╯  (empty) prepare to deploy to the cloud
│ │ ◉ │  yxxppztp steve@steveklabnik.com 2024-03-17 14:18:00.000
-05:00 push-yxxppztpoyqq 3d123151
│ │ │ │  (empty) have galactus query eks with time range
<snip>
~
```

We're back to our beautiful tree, though our `@` working commit got lost in the shuffle. It was empty anyway! Let's bring it back:

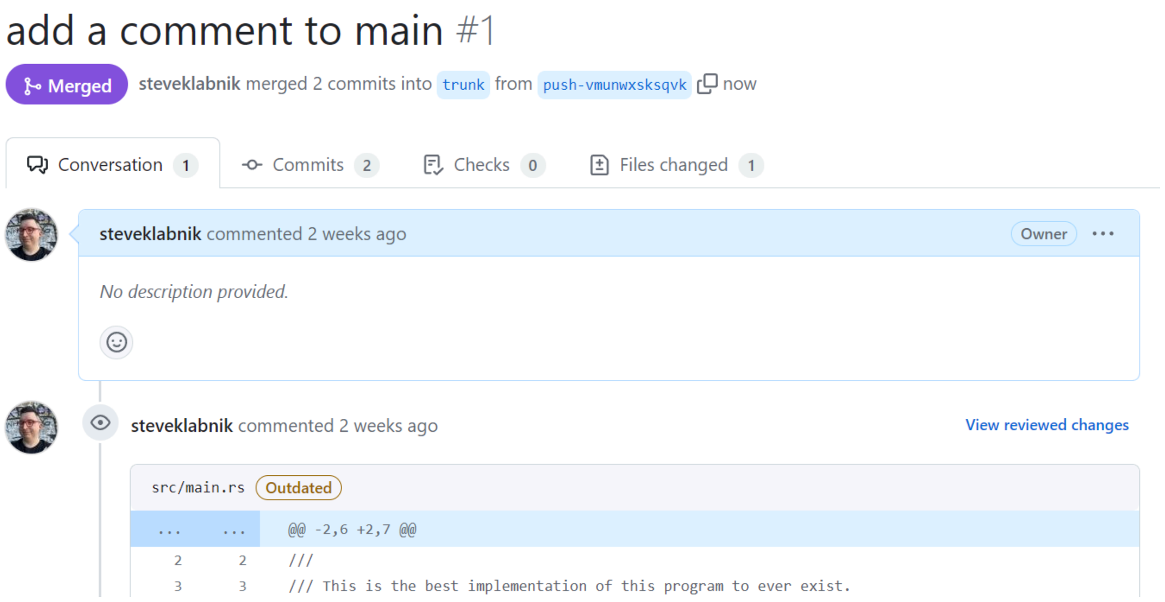
```
> jj new xn
Working copy now at: pptrunzw 06442487 (empty) (no description set)
Parent commit      : xnutwmso bcf8a74b (empty) merge: steve's branch
Added 0 files, modified 2 files, removed 0 files
> jj log
@  pptrunzw steve@steveklabnik.com 2024-03-17 14:55:59.000 -05:00
06442487
│  (empty) (no description set)
◉         xnutwmso steve@steveklabnik.com 2024-03-17 14:52:08.000
-05:00 bcf8a74b
├─┬─┬─┬─╮  (empty) merge: steve's branch

<snip>
```

Excellent.

Rebasing on upstream changes

What happens when some changes land upstream, and we need to rebase our pull requests. If you're like me, you often have multiple PRs going at any given time. Rebasing them all can be very tedius. Here's how we can do that with this workflow.

First, let's make a change upstream. I'm going to do this by merging my open PR from earlier in the tutorial:



add a comment to main #1

Merged steveklabnik merged 2 commits into `trunk` from `push-vmunwxsksqvk` now

Conversation 1 · Commits 2 · Checks 0 · Files changed 1

steveklabnik commented 2 weeks ago · Owner

No description provided.

steveklabnik commented 2 weeks ago · View reviewed changes

src/main.rs Outdated

```
...   ...   @@ -2,6 +2,7 @@
 2     2    ///
 3     3    /// This is the best implementation of this program to ever exist.
```

After fetching changes, our log looks like this:

```
> jj git fetch
> jj log
@  xqkmpxlq steve@steveklabnik.com 2024-03-17 15:18:11.000 -05:00
fccf0626
│  (empty) (no description set)
◉         xnutwmso steve@steveklabnik.com 2024-03-17 15:16:36.000
-05:00 da67dfe1
├─┬─┬─┬─╮  (empty) merge: steve's branch
│ │ │ │ ◉  kvupxvpv steve@steveklabnik.com 2024-03-17 15:15:20.000
-05:00 2ea49586
│ │ │ │ │  (empty) second 80% done
│ │ │ │ ◉  zxyukunn steve@steveklabnik.com 2024-03-17 14:24:56.000
-05:00 push-zxyukunnwolo 30081a6b
│ │ │ │ │  (empty) first 80% done
│ │ │ │ ◉  tzsloruo steve@steveklabnik.com 2024-03-17 14:24:21.000
-05:00 7c02f6ce
│ │ │ │ │  (empty) another feature
│ │ │ ◉ │  rxpztwms steve@steveklabnik.com 2024-03-17 14:23:00.000
-05:00 push-rxpztwmsszvk 902a6cd2
│ │ │ │ │  (empty) various fixes
│ │ │ ◉ │  tmnmvxyy steve@steveklabnik.com 2024-03-17 14:22:15.000
-05:00 105cf6b5
│ │ │ ├─╯  (empty) prepare to deploy to the cloud
│ │ ◉ │  yxxppztp steve@steveklabnik.com 2024-03-17 14:18:00.000
-05:00 push-yxxppztpoyqq 3d123151
│ │ │ │  (empty) have galactus query eks with time range
│ │ ◉ │  opwqpunl steve@steveklabnik.com 2024-03-17 14:15:58.000
-05:00 1a66beb1
│ │ ├─╯  (empty) display the birthday date on the settings page
│ ◉ │  ymvptyyq steve@steveklabnik.com 2024-03-17 14:25:31.000
-05:00 push-ymvptyyqmyul 728dbb1e
│ │ │  (empty) fixing all the breakage from updating dependencies
│ ◉ │  xulymzyp steve@steveklabnik.com 2024-03-17 14:25:14.000
-05:00 1f7c69a5
│ ├─╯  (empty) updating dependencies
◉ │  msmntwvo?? steve@steveklabnik.com 2024-03-17 14:45:41.000
-05:00 push-vmunwxsksqvk* cdca9211
│ │  add a new function
◉ │  vmunwxsk?? steve@steveklabnik.com 2024-03-17 14:45:41.000
-05:00 3a08be8a
├─╯  add a comment to main
│ ◉  okyzuxnk steve@steveklabnik.com 2024-03-17 14:59:02.000 -05:00
trunk b7f9d708
╭─╯  (empty) Merge pull request #1 from steveklabnik/push-
vmunwxsksqvk
◉  ksrmwuon steve@steveklabnik.com 2024-03-01 23:10:35.000 -06:00
e202b67c
```

```
│  Update Cargo.toml
~
```

Uh oh! That's got some concerning stuff. The changes with `??` and are in red, and there's also the new merge commit that `trunk` is set up to. Since all of our branches were off of the change where `trunk` used to be, we should rebase them on top of the new trunk. First, we want to make sure that `@` is at the empty change on top of our merge commit. You can see from the `jj log` just above that that is true.

We can rebase all of our PRs with one command:

```
> jj rebase -s 'all:roots(trunk..@)' -d trunk
Rebased 14 commits
Working copy now at: ltupzukw 9a496ef6 (empty) (no description set)
Parent commit      : xnutwmso 6be25a32 (empty) merge: steve's branch
```

This is using some revset stuff we haven't seen before! Let's break it down:

This is using `jj rebase -s` rather than `-r`, like we've been doing before. Here's the description from `jj rebase --help`:

```
  -s, --source <SOURCE>
          Rebase specified revision(s) together their tree of
descendants (can be repeated)
```

That's a little rough. There are some helpful diagrams in `jj rebase --help` that got me to understand the differences between `-r`, `-s`, and `-b`, but the short of it is that `-r` will sort of "rip out" a change and move it somewhere else. `-s` does that, but also moves descendants. In other words, imagine that we have a history like this:

```
A - B - C - D
```

If we `jj rebase -r C` to somewhere else, it will only move that revision, and so you end up with

```
A - B - D
```

Whereas `-s` operates more like I'd be used to with `git`: it takes the children too, so after `jj rebase -s C`, you'd have:

```
A - B
```

as both `C` and `D` are somewhere else now. `-b` works to rebase a branch.

The next part, `all:' is a prefix. What's the prefix do? Well, let's try running the command without it:

```
> jj rebase -s 'roots(trunk..@)' -d trunk
Error: Revset "roots(trunk..@)" resolved to more than one revision
Hint: The revset "roots(trunk..@)" resolved to these revisions:
opwqpunl ba100a96 (empty) display the birthday date on the settings
page
tmnmvxyy 462dcf72 (empty) prepare to deploy to the cloud
tzsloruo fdba6abd (empty) another feature
xulymzyp abbf424e (empty) updating dependencies
vmunwxsk?? 2ce46bd0 (empty) add a comment to main
Prefix the expression with 'all' to allow any number of revisions
(i.e. 'all:roots(trunk..@)').
```

This is basically a way to help make sure you've got the right arguments: sometimes when working with revsets, you expect the result to be only one revision, and sometimes you expect it to be many revisions. For some commands, one or the other may be not necessarily what you want. In this case, most of the time, when you rebase, you only want one parent. So if we use a revset that returns more than one change, that might be a bug! So `jj rebase` wants us to reassure it that we are creating a change with multiple parents by putting `all:` as a prefix.

Finally, `trunk..@` is being passed to the `roots()` function. `trunk..@` is a range, so it will give every change between where `trunk` is and `@`. The `roots()` function gives back the roots of the changes provided to it, so in this case, commits that have children but no parents. To which you may say, "but don't those commits have parents?" In general, sure, but because we are only returning the set of those commits, their parents aren't in the set, and therefore don't exist for the purposes of a function like this. This means we end up with the commits that have `trunk` as parents, and have children themselves. The roots of the tree.

Let's put it all together:

```
$ jj rebase -s 'all:roots(trunk..@)' -d trunk
```

We're rebasing all of the root changes from `trunk` to `@` onto `trunk`.

Kind of a mouthful, but not too bad to understand!

Here's what our `jj log` looks like:

```
> jj log
◉ xnutwmso steve@steveklabnik.com 2024-03-17 16:01:56.000 -05:00
ce833ae7
│ (empty) merge: steve's branch
◉     xqkmpxlq steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 5dc292c2
├┬┬┬╮ (empty) (no description set)
│││││@ ltupzukw steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 edf9cd58
╭┬┬┬┬╯ (empty) (no description set)
││││◉ yxxppztp steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 push-yxxppztpoyqq* b3db74d3
│││││ (empty) have galactus query eks with time range
││││◉ opwqpunl steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 713c692d
│││││ (empty) display the birthday date on the settings page
│││◉│ rxpztwms steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 push-rxpztwmsszvk* 76dbbcd1
│││││ (empty) various fixes
│││◉│ tmnmvxyy steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 d0f7f627
│││├╯ (empty) prepare to deploy to the cloud
││◉│ ymvptyyq steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 push-ymvptyyqmyul* f448e93a
││││ (empty) fixing all the breakage from updating dependencies
││◉│ xulymzyp steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 d608ebd3
││├╯ (empty) updating dependencies
│◉│ msmntwvo?? steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 push-vmunwxsksqvk* 21569f7a
│││ (empty) add a new function
│◉│ vmunwxsk?? steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 9a050939
│├╯ (empty) add a comment to main
◉│ kvupxvpv steve@steveklabnik.com 2024-03-17 16:01:56.000 -05:00
3b0a722a
││ (empty) second 80% done
◉│ zxyukunn steve@steveklabnik.com 2024-03-17 16:01:56.000 -05:00
push-zxyukunnwolo* 7e826ad4
││ (empty) first 80% done
◉│ tzsloruo steve@steveklabnik.com 2024-03-17 16:01:56.000 -05:00
792cc601
├╯ (empty) another feature
◉ okyzuxnk steve@steveklabnik.com 2024-03-17 14:59:02.000 -05:00
trunk b7f9d708
```

```
│  (empty) Merge pull request #1 from steveklabnik/push-vmunwxsksqvk
~
```

Let's move `@` back to where we want it:

```
> jj new xn
Working copy now at: vvvouunp 78919d69 (empty) (no description set)
Parent commit      : xnutwmso ce833ae7 (empty) merge: steve's branch
> jj abandon l
Abandoned commit ltupzukw edf9cd58 (empty) (no description set)
> jj log
@  vvvouunp steve@steveklabnik.com 2024-03-17 16:02:39.000 -05:00
78919d69
│  (empty) (no description set)
◉  xnutwmso steve@steveklabnik.com 2024-03-17 16:01:56.000 -05:00
ce833ae7
│  (empty) merge: steve's branch
◉          xqkmpxlq steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 5dc292c2
├─┬─┬─┬─╮  (empty) (no description set)
│ │ │ │ ◉  yxxppztp steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 push-yxxppztpoyqq* b3db74d3
│ │ │ │ │  (empty) have galactus query eks with time range
│ │ │ │ ◉  opwqpunl steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 713c692d
│ │ │ │ │  (empty) display the birthday date on the settings page
│ │ │ ◉ │  rxpztwms steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 push-rxpztwmsszvk* 76dbbcd1
│ │ │ │ │  (empty) various fixes
│ │ │ ◉ │  tmnmvxyy steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 d0f7f627
│ │ │ ├─╯  (empty) prepare to deploy to the cloud
│ │ ◉ │  ymvptyyq steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 push-ymvptyyqmyul* f448e93a
│ │ │ │  (empty) fixing all the breakage from updating dependencies
│ │ ◉ │  xulymzyp steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 d608ebd3
│ │ ├─╯  (empty) updating dependencies
│ ◉ │  msmntwvo?? steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 push-vmunwxsksqvk* 21569f7a
│ │ │  (empty) add a new function
│ ◉ │  vmunwxsk?? steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 9a050939
│ ├─╯  (empty) add a comment to main
◉ │  kvupxvpv steve@steveklabnik.com 2024-03-17 16:01:56.000 -05:00
3b0a722a
│ │  (empty) second 80% done
◉ │  zxyukunn steve@steveklabnik.com 2024-03-17 16:01:56.000 -05:00
push-zxyukunnwolo* 7e826ad4
│ │  (empty) first 80% done
◉ │  tzsloruo steve@steveklabnik.com 2024-03-17 16:01:56.000 -05:00
792cc601
```

```
├─╯  (empty) another feature
◉  okyzuxnk steve@steveklabnik.com 2024-03-17 14:59:02.000 -05:00
trunk b7f9d708
│  (empty) Merge pull request #1 from steveklabnik/push-vmunwxsksqvk
~
```

Looking pretty good! One last thing: what's up with those red commits? Well, those are from our external PR, and we don't need them. So we can just abandon that branch:

```
> jj abandon push-vmunwxsksqvk
Abandoned commit msmntwvo?? 21569f7a push-vmunwxsksqvk* | (empty)
add a new function
Rebased 3 descendant commits onto parents of abandoned commits
Working copy now at: vvvouunp ace3f1a2 (empty) (no description set)
Parent commit      : xnutwmso 32871810 (empty) merge: steve's branch
> jj abandon push-vmunwxsksqvk
Abandoned commit vmunwxsk?? 9a050939 push-vmunwxsksqvk* | (empty)
add a comment to main
Rebased 3 descendant commits onto parents of abandoned commits
Working copy now at: vvvouunp e3f9254f (empty) (no description set)
Parent commit      : xnutwmso 0459bd1c (empty) merge: steve's branch
```

And now our log is clean:

```
> jj log
@  vvvouunp steve@steveklabnik.com 2024-03-17 16:06:15.000 -05:00
e3f9254f
│  (empty) (no description set)
◉  xnutwmso steve@steveklabnik.com 2024-03-17 16:06:15.000 -05:00
0459bd1c
│  (empty) merge: steve's branch
◉        xqkmpxlq steve@steveklabnik.com 2024-03-17 16:06:15.000
-05:00 c9e5fa35
├─┬─┬─╮  (empty) (no description set)
│ │ │ ◉  yxxppztp steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 push-yxxppztpoyqq* b3db74d3
│ │ │ │  (empty) have galactus query eks with time range
│ │ │ ◉  opwqpunl steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 713c692d
│ │ │ │  (empty) display the birthday date on the settings page
│ │ ◉ │  rxpztwms steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 push-rxpztwmsszvk* 76dbbcd1
│ │ │ │  (empty) various fixes
│ │ ◉ │  tmnmvxyy steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 d0f7f627
│ │ ├─╯  (empty) prepare to deploy to the cloud
│ ◉ │  ymvptyyq steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 push-ymvptyyqmyul* f448e93a
│ │ │  (empty) fixing all the breakage from updating dependencies
│ ◉ │  xulymzyp steve@steveklabnik.com 2024-03-17 16:01:56.000
-05:00 d608ebd3
│ ├─╯  (empty) updating dependencies
◉ │  kvupxvpv steve@steveklabnik.com 2024-03-17 16:01:56.000 -05:00
3b0a722a
│ │  (empty) second 80% done
◉ │  zxyukunn steve@steveklabnik.com 2024-03-17 16:01:56.000 -05:00
push-zxyukunnwolo* 7e826ad4
│ │  (empty) first 80% done
◉ │  tzsloruo steve@steveklabnik.com 2024-03-17 16:01:56.000 -05:00
792cc601
├─╯  (empty) another feature
◉  okyzuxnk steve@steveklabnik.com 2024-03-17 14:59:02.000 -05:00
push-vmunwxsksqvk* trunk b7f9d708
│  (empty) Merge pull request #1 from steveklabnik/push-vmunwxsksqvk
~
```

And there we go!

Conclusion

This workflow may not make sense to you, but it is a very neat example of using the tools you already know to do something completely different! We're going to talk about a slightly different workflow next: "stacked pull requests". They're similar in some ways to this approach, but there's also some differences too.

Customizing the output of various `jj` commands with templates

Just like revsets are a functional language that allow you to describe a set of commits, templates are a typed functional language that allows you to customize the output of commands in `jj`.

A bunch of commands support `-T` or `--template` to allow you to customize their output. For example, right now `@` is on an empty commit, so let's refresh ourselves on what

```
$ jj log -r @-
◉  yzlysuwt steve@steveklabnik.com 2024-02-29 00:35:12.000 -06:00
main f80a73c1
│  Fill out a table of contents
```

Let's give it a template instead:

```
$  jj log -r @- -T 'separate(" ", change_id,
description.first_line())'
◉  yzlysuwtylswszxknppsyqxqoktqpqpz Fill out a table of contents
│
```

That's quite different! We've got a few things going on here:

- `separate()` is a function that takes a separator, a bunch of other templates, and produces the contents of those templates separated by the separator.
- `change_id` and `description` are keywords that resolve to what you'd expect.
- `first_line()` is a method on strings that returns the first line.

What if we wanted a nicer change ID? We can do that:

```
$  jj log -r @- -T 'separate(" ", change_id.shortest(8),
description.first_line())'
◉  yzlysuwt Fill out a table of contents
│
```

The `shortest` method returns the shortest prefix of an ID, and the `8` we pass to it says "show at least eight letters even if the shortest is shorter."

Templates are powerful, and let you do a lot of interesting things. I would suggest reading the documentation on templates to learn all of the details.