MyFirstContribution

- Summary
  - Prerequisites
  - Related Reading
  - Getting Help
    - git@vger.kernel.org
    - #git-devel on Libera Chat
    - #discord on Discord
- Getting Started
  - Clone the Git Repository
  - Installing Dependencies
  - Identify Problem to Solve
  - Set Up Your Workspace
- Code It Up!
  - Adding a New Command
  - Implementation
  - Adding Documentation
  - Adding Usage Text
- Testing
  - Overview of Testing Structure
  - Writing Your Test
  - Running Locally
- Getting Ready to Share: Anatomy of a Patch Series
  - The cover letter
- Sending Patches via GitGitGadget
  - Forking git/git on GitHub
  - Uploading to Your Own Fork
  - Sending a PR to GitGitGadget
  - Running CI and Getting Ready to Send

    - Sending Your Patches
    - Updating With Comments
- Sending Patches with git send-email
    - Prerequisite: Setting Up git send-email
    - Preparing Initial Patchset
    - Preparing Email
    - Sending Email
    - Sending v2
    - Bonus Chapter: One-Patch Changes
- Sending Patches with b4
    - Preparing a Patch Series
    - Sending the Patches
    - Sending v2
    - Configure b4
- My Patch Got Emailed - Now What?
    - Responding to Reviews
    - After Review Approval

My First Contribution to the Git Project
========== :sectanchors:

Summary

This is a tutorial demonstrating the end-to-end workflow of creating a change to the Git tree, sending it for review, and making changes based on comments.

Prerequisites

This tutorial assumes you’re already fairly familiar with using Git to manage source code. The Git workflow steps will largely remain unexplained.

Related Reading

This tutorial aims to summarize the following documents, but the reader may find useful additional context:

- `Documentation/SubmittingPatches`
- `Documentation/howto/new-command.adoc`

Getting Help

If you get stuck, you can seek help in the following places.

git@vger.kernel.org

This is the main Git project mailing list where code reviews, version announcements, design discussions, and more take place. Those interested in contributing are welcome to post questions here. The Git list requires plain-text-only emails and prefers inline and bottom-posting when replying to mail; you will be CC'd in all replies to you. Optionally, you can subscribe to the list by sending an email to <git+subscribe@vger.kernel.org> (see https://subspace.kernel.org/subscribing.html for details). The archive of this mailing list is available to view in a browser.

#git-devel on Libera Chat

This IRC channel is for conversations between Git contributors. If someone is currently online and knows the answer to your question, you can receive help in real time. Otherwise, you can read the scrollback to see whether someone answered you. IRC does not allow offline private messaging, so if you try to private message someone and then log out of IRC, they cannot respond to you. It's better to ask your questions in the channel so that you can be answered if you disconnect and so that others can learn from the conversation.

#discord on Discord

This is an unofficial Git Discord server for everyone, from people just starting out with Git to those who develop it. It's a great place to ask questions, share tips, and connect with the broader Git community in real time.

The server has channels for general discussions and specific channels for those who use Git and those who develop it. The server's search functionality also allows you to find previous conversations and answers to common questions.

Getting Started

Clone the Git Repository

Git is mirrored in a number of locations. Clone the repository from one of them; https://git-scm.com/downloads suggests one of the best places to clone from is the mirror on GitHub.

```
$ git clone https://github.com/git/git git
$ cd git
```

Installing Dependencies

To build Git from source, you need to have a handful of dependencies installed on your system. For a hint of what's needed, you can take a look at `INSTALL`, paying close attention to the section about Git's dependencies on external programs and libraries. That document mentions a way to "test-drive" our freshly built Git without installing; that's the method we'll be using in this tutorial.

Make sure that your environment has everything you need by building your brand new clone of Git from the above step:

```
$ make
```

> [!NOTE] The Git build is parallelizable. `-j#` is not included above but you can use it as you prefer, here and elsewhere.

Identify Problem to Solve

In this tutorial, we will add a new command, git psuh, short for ``Pony Saying `Um, Hello''' - a feature which has gone unimplemented despite a high frequency of invocation during users' typical daily workflow.

(We've seen some other effort in this space with the implementation of popular commands such as `sl`.)

Set Up Your Workspace

Let's start by making a development branch to work on our changes. Per `Documentation/SubmittingPatches`, since a brand new command is a new feature, it's fine to base your work on `master`. However, in the future for bugfixes, etc., you should check that document and base it on the appropriate branch.

For the purposes of this document, we will base all our work on the `master` branch of the upstream project. Create the `psuh` branch you will use for development like so:

```
$ git checkout -b psuh origin/master
```

We'll make a number of commits here in order to demonstrate how to send a topic with multiple patches up for review simultaneously.

Code It Up!

> [!NOTE] A reference implementation can be found at https://github.com/nasamuffin/git/tree/psuh.

Adding a New Command

Lots of the subcommands are written as builtins, which means they are implemented in C and compiled into the main `git` executable. Implementing the very simple `psuh` command as a built-in will demonstrate the structure of the codebase, the internal API, and the process of working together as a contributor with the reviewers and maintainer to integrate this change into the system.

Built-in subcommands are typically implemented in a function named "cmd_" followed by the name of the subcommand, in a source file named after the subcommand and contained within `builtin/`. So it makes sense to implement your command in `builtin/psuh.c`. Create that file, and within it, write the entry point for your command in a function matching the style and signature:

```
int cmd_psuh(int argc UNUSED, const char **argv UNUSED,
         const char *prefix UNUSED, struct repository *repo UNUSED)
```

A few things to note:

- A subcommand implementation takes its command line arguments in `int` `argc` + `const` `char` `**argv`, like `main()` would.
- It also takes two extra parameters, `prefix` and `repo`. What they mean will not be discussed until much later.
- Because this first example will not use any of the parameters, your compiler will give warnings on unused parameters. As the list of these four parameters is mandated by the API to add new built-in commands, you cannot omit them. Instead, you add `UNUSED` to each of them to tell the compiler that you **know** you are not (yet) using it.

We'll also need to add the declaration of psuh; open up `builtin.h`, find the declaration for `cmd_pull`, and add a new line for `psuh` immediately before it, in order to keep the declarations alphabetically sorted:

```
int cmd_psuh(int argc, const char **argv, const char *prefix, struct repository *repo);
```

Be sure to `#include` `"builtin.h"` in your `psuh.c`. You'll also need to `#include` `"gettext.h"` to use functions related to printing output text.

Go ahead and add some throwaway printf to the `cmd_psuh` function. This is a decent starting point as we can now add build rules and register the command.

> [!NOTE] Your throwaway text, as well as much of the text you will be adding over the course of this tutorial, is user-facing. That means it needs to be localizable. Take a look at `po/README` under "Marking strings for translation". Throughout the tutorial, we will mark strings for translation as necessary; you should also do so when writing your user-facing commands in the future.

```
int cmd_psuh(int argc UNUSED, const char **argv UNUSED,
         const char *prefix UNUSED, struct repository *repo UNUSED)
{
    printf(_("Pony saying hello goes here.\n"));
    return 0;
}
```

Let's try to build it. Open `Makefile`, find where `builtin/pull.o` is added to `BUILTIN_OBJS`, and add `builtin/psuh.o` in the same way next to it in alphabetical order. Once you've done so, move to the top-level directory and build simply with `make`. Also add the `DEVELOPER=1` variable to turn on some additional warnings:

```
$ echo DEVELOPER=1 >config.mak
$ make
```

> [!NOTE] When you are developing the Git project, it's preferred that you use the `DEVELOPER` flag; if there's some reason it doesn't work for you, you can turn it off, but it's a good idea to mention the problem to the mailing list.

Great, now your new command builds happily on its own. But nobody invokes it. Let's change that.

The list of commands lives in `git.c`. We can register a new command by adding a `cmd_struct` to the `commands[]` array. `struct` `cmd_struct` takes a string with the command name, a function pointer to the command implementation, and a setup option flag. For now, let's keep mimicking `push`. Find the line where `cmd_push` is registered, copy it, and modify it for `cmd_psuh`, placing the new line in alphabetical order (immediately before `cmd_pull`).

The options are documented in `builtin.h` under "Adding a new built-in." Since we hope to print some data about the user's current workspace context later, we need a Git directory, so choose `RUN_SETUP` as your only option.

Go ahead and build again. You should see a clean build, so let's kick the tires and see if it works. There's a binary you can use to test with in the `bin-wrappers` directory.

```
$ ./bin-wrappers/git psuh
```

Check it out! You've got a command! Nice work! Let's commit this.

`git` `status` reveals modified `Makefile`, `builtin.h`, and `git.c` as well as untracked `builtin/psuh.c` and `git-psuh`. First, let's take care of the binary, which should be ignored. Open `.gitignore` in your editor, find `/git-pull`, and add an entry for your new command in alphabetical order:

```
...
/git-prune-packed
/git-psuh
/git-pull
/git-push
/git-quiltimport
/git-range-diff
```

```
...
```

Checking `git status` again should show that `git-psuh` has been removed from the untracked list and `.gitignore` has been added to the modified list. Now we can stage and commit:

```
$ git add Makefile builtin.h builtin/psuh.c git.c .gitignore
$ git commit -s
```

You will be presented with your editor in order to write a commit message. Start the commit with a 50-column or less subject line, including the name of the component you're working on, followed by a blank line (always required) and then the body of your commit message, which should provide the bulk of the context. Remember to be explicit and provide the "Why" of your change, especially if it couldn't easily be understood from your diff. When editing your commit message, don't remove the `Signed-off-by` trailer which was added by `-s` above.

```
psuh: add a built-in by popular demand

Internal metrics indicate this is a command many users expect to be
present. So here's an implementation to help drive customer
satisfaction and engagement: a pony which doubtfully greets the user,
or, a Pony Saying "Um, Hello" (PSUH).

This commit message is intentionally formatted to 72 columns per line,
starts with a single line as "commit message subject" that is written as
if to command the codebase to do something (add this, teach a command
that). The body of the message is designed to add information about the
commit that is not readily deduced from reading the associated diff,
such as answering the question "why?".

Signed-off-by: A U Thor <author@example.com>
```

Go ahead and inspect your new commit with `git show`. "psuh:" indicates you have modified mainly the `psuh` command. The subject line gives readers an idea of what you've changed. The sign-off line (`-s`) indicates that you agree to the Developer's Certificate of Origin 1.1 (see the `Documentation/SubmittingPatches` [[dco]] header).

For the remainder of the tutorial, the subject line only will be listed for the sake of brevity. However, fully-fleshed example commit messages are available on the reference implementation linked at the top of this document.

Implementation

It’s probably useful to do at least something besides printing out a string. Let’s start by having a look at everything we get.

Modify your `cmd_psuh` implementation to dump the args you’re passed, keeping existing `printf()` calls in place; because the args are now used, remove the `UNUSED` macro from them:

```
    int i;

    ...

    printf(Q_("Your args (there is %d):\n",
          "Your args (there are %d):\n",
          argc),
           argc);
    for (i = 0; i < argc; i++)
        printf("%d: %s\n", i, argv[i]);

    printf(_("Your current working directory:\n<top-level>%s%s\n"),
           prefix ? "/" : "", prefix ? prefix : "");
```

Build and try it. As you may expect, there’s pretty much just whatever we give on the command line, including the name of our command. (If `prefix` is empty for you, try `cd Documentation/ && ../bin-wrappers/git psuh`). That’s not so helpful. So what other context can we get?

Add a line to `#include "config.h"`, `#include "repository.h"` and `#include "environment.h"`. Then, add the following bits to the function body: function body:

```
    const char *cfg_name;

...

    repo_config(repo, git_default_config, NULL);
    if (repo_config_get_string_tmp(repo, "user.name", &cfg_name))
        printf(_("No name is found in config\n"));
    else
        printf(_("Your name: %s\n"), cfg_name);
```

`repo_config()` will grab the configuration from config files known to Git and apply standard precedence rules. `repo_config_get_string_tmp()` will look up a specific key ("user.name") and give you the value. There are a number of single-key lookup functions like this one; you can see them all (and more info about how to use `repo_config()`) in `config.h`.

You should see that the name printed matches the one you see when you run:

```
$ git config --get user.name
```

Great! Now we know how to check for values in the Git config. Let's commit this too, so we don't lose our progress.

```
$ git add builtin/psuh.c
$ git commit -sm "psuh: show parameters & config opts"
```

> [!NOTE] Again, the above is for sake of brevity in this tutorial. In a real change you should not use `-m` but instead use the editor to write a meaningful message.

Still, it'd be nice to know what the user's working context is like. Let's see if we can print the name of the user's current branch. We can mimic the `git status` implementation; the printer is located in `wt-status.c` and we can see that the branch is held in a `struct wt_status`.

`wt_status_print()` gets invoked by `cmd_status()` in `builtin/commit.c`. Looking at that implementation we see the status config being populated like so:

```
status_init_config(&s, git_status_config);
```

But as we drill down, we can find that `status_init_config()` wraps a call to `repo_config()`. Let's modify the code we wrote in the previous commit.

Be sure to include the header to allow you to use `struct wt_status`:

```
#include "wt-status.h"
```

Then modify your `cmd_psuh` implementation to declare your `struct wt_status`, prepare it, and print its contents:

```
    struct wt_status status;

...

    wt_status_prepare(repo, &status);
    repo_config(repo, git_default_config, &status);

...

    printf(_("Your current branch: %s\n"), status.branch);
```

Run it again. Check it out - here's the (verbose) name of your current branch!

Let's commit this as well.

```
$ git add builtin/psuh.c
$ git commit -sm "psuh: print the current branch"
```

Now let's see if we can get some info about a specific commit.

Luckily, there are some helpers for us here. `commit.h` has a function called `lookup_commit_reference_by_name` to which we can simply provide a hardcoded string; `pretty.h` has an extremely handy `pp_commit_easy()` call which doesn't require a full format object to be passed.

Add the following includes:

```
#include "commit.h"
#include "pretty.h"
#include "strbuf.h"
```

Then, add the following lines within your implementation of `cmd_psuh()` near the declarations and the logic, respectively.

```
    struct commit *c = NULL;
    struct strbuf commitline = STRBUF_INIT;

...

    c = lookup_commit_reference_by_name("origin/master");

    if (c != NULL) {
        pp_commit_easy(CMIT_FMT_ONELINE, c, &commitline);
        printf(_("Current commit: %s\n"), commitline.buf);
    }
```

The `struct` `strbuf` provides some safety belts to your basic `char*`, one of which is a length member to prevent buffer overruns. It needs to be initialized nicely with `STRBUF_INIT`. Keep it in mind when you need to pass around `char*`.

`lookup_commit_reference_by_name` resolves the name you pass it, so you can play with the value there and see what kind of things you can come up with.

`pp_commit_easy` is a convenience wrapper in `pretty.h` that takes a single format enum shorthand, rather than an entire format struct. It then pretty-prints the commit according to that shorthand. These are similar to the formats available with `--pretty=FOO` in many Git commands.

Build it and run, and if you're using the same name in the example, you should see the subject line of the most recent commit in `origin/master` that you know about. Neat! Let's commit that as well.

```
$ git add builtin/psuh.c
$ git commit -sm "psuh: display the top of origin/master"
```

Adding Documentation

Awesome! You've got a fantastic new command that you're ready to share with the community. But hang on just a minute - this isn't very user-friendly. Run the following:

```
$ ./bin-wrappers/git help psuh
```

Your new command is undocumented! Let's fix that.

Take a look at `Documentation/git-*.adoc`. These are the manpages for the subcommands that Git knows about. You can open these up and take a look to get acquainted with the format, but then go ahead and make a new file `Documentation/git-psuh.adoc`. Like with most of the documentation in the Git project, help pages are written with AsciiDoc (see CodingGuidelines, "Writing Documentation" section). Use the following template to fill out your own manpage:

```
git-psuh(1)
===========

NAME
----
git-psuh - Delight users' typo with a shy horse



SYNOPSIS
--------
[synopsis]
git psuh [<arg>...]

DESCRIPTION
-----------
...

OPTIONS[[OPTIONS]]
------------------
```

```
...

OUTPUT
------
...

GIT
---
Part of the linkgit:git[1] suite
```

The most important pieces of this to note are the file header, underlined by =, the NAME section, and the SYNOPSIS, which would normally contain the grammar if your command took arguments. Try to use well-established manpage headers so your documentation is consistent with other Git and UNIX manpages; this makes life easier for your user, who can skip to the section they know contains the information they need.

> [!NOTE] Before trying to build the docs, make sure you have the package `asciidoc` installed.

Now that you've written your manpage, you'll need to build it explicitly. We convert your AsciiDoc to troff which is man-readable like so:

```
$ make all doc
$ man Documentation/git-psuh.1
```

or

```
$ make -C Documentation/ git-psuh.1
$ man Documentation/git-psuh.1
```

While this isn't as satisfying as running through `git help`, you can at least check that your help page looks right.

You can also check that the documentation coverage is good (that is, the project sees that your command has been implemented as well as documented) by running `make check-docs` from the top-level.

Go ahead and commit your new documentation change.

Adding Usage Text

Try and run `./bin-wrappers/git` `psuh` `-h`. Your command should crash at the end. That's because `-h` is a special case which your command should handle by printing usage.

Take a look at `Documentation/technical/api-parse-options.adoc`. This is a handy tool for pulling out options you need to be able to handle, and it takes a usage string.

In order to use it, we'll need to prepare a NULL-terminated array of usage strings and a `builtin_psuh_options` array.

Add a line to `#include` `"parse-options.h"`.

At global scope, add your array of usage strings:

```
static const char * const psuh_usage[] = {
    N_("git psuh [<arg>...]"),
    NULL,
};
```

Then, within your `cmd_psuh()` implementation, we can declare and populate our `option` struct. Ours is pretty boring but you can add more to it if you want to explore `parse_options()` in more detail:

```
    struct option options[] = {
        OPT_END()
    };
```

Finally, before you print your args and prefix, add the call to `parse-options()`:

```
    argc = parse_options(argc, argv, prefix, options, psuh_usage, 0);
```

This call will modify your `argv` parameter. It will strip the options you specified in `options` from `argv` and the locations pointed to from `options` entries will be updated. Be sure to replace your `argc` with the result from `parse_options()`, or you will be confused if you try to parse `argv` later.

It's worth noting the special argument `--`. As you may be aware, many Unix commands use `--` to indicate "end of named parameters" - all parameters after the `--` are interpreted merely as positional arguments. (This can be handy if you want to pass as a parameter something which would usually be interpreted as a flag.) `parse_options()` will terminate parsing when it reaches `--` and give you the rest of the options afterwards, untouched.

Now that you have a usage hint, you can teach Git how to show it in the general command list shown by `git` `help` `git` or `git` `help` `-a`, which is generated from `command-list.txt`. Find the line

for ’git-pull’ so you can add your ’git-psuh’ line above it in alphabetical order. Now, we can add some attributes about the command which impacts where it shows up in the aforementioned help commands. The top of `command-list.txt` shares some information about what each attribute means; in those help pages, the commands are sorted according to these attributes. `git psuh` is user-facing, or porcelain - so we will mark it as "mainporcelain". For "mainporcelain" commands, the comments at the top of `command-list.txt` indicate we can also optionally add an attribute from another list; since `git psuh` shows some information about the user’s workspace but doesn’t modify anything, let’s mark it as "info". Make sure to keep your attributes in the same style as the rest of `command-list.txt` using spaces to align and delineate them:

```
git-prune-packed                        plumbingmanipulators
git-psuh                                mainporcelain       info
git-pull                                mainporcelain           remote
git-push                                mainporcelain           remote
```

Build again. Now, when you run with `-h`, you should see your usage printed and your command terminated before anything else interesting happens. Great!

Go ahead and commit this one, too.

Testing

It’s important to test your code - even for a little toy command like this one. Moreover, your patch won’t be accepted into the Git tree without tests. Your tests should:

- Illustrate the current behavior of the feature
- Prove the current behavior matches the expected behavior
- Ensure the externally-visible behavior isn’t broken in later changes

So let’s write some tests.

Related reading: `t/README`

Overview of Testing Structure

The tests in Git live in `t/` and are named with a 4-digit decimal number using the schema shown in the Naming Tests section of `t/README`.

Writing Your Test

Since this a toy command, let's go ahead and name the test with t9999. However, as many of the family/subcmd combinations are full, best practice seems to be to find a command close enough to the one you've added and share its naming space.

Create a new file `t/t9999-psuh-tutorial.sh`. Begin with the header as so (see "Writing Tests" and "Source 'test-lib.sh'" in `t/README`):

```
#!/bin/sh

test_description='git-psuh test

This test runs git-psuh and makes sure it does not crash.'

. ./test-lib.sh
```

Tests are framed inside of a `test_expect_success` in order to output TAP formatted results. Let's make sure that `git` `psuh` doesn't exit poorly and does mention the right animal somewhere:

```
test_expect_success 'runs correctly with no args and good output' '
    git psuh >actual &&
    grep Pony actual
'
```

Indicate that you've run everything you wanted by adding the following at the bottom of your script:

```
test_done
```

Make sure you mark your test script executable:

```
$ chmod +x t/t9999-psuh-tutorial.sh
```

You can get an idea of whether you created your new test script successfully by running `make` `-C` `t` `test-lint`, which will check for things like test number uniqueness, executable bit, and so on.

Running Locally

Let's try and run locally:

```
$ make
$ cd t/ && prove t9999-psuh-tutorial.sh
```

You can run the full test suite and ensure `git-psuh` didn't break anything:

```
$ cd t/
$ prove -j$(nproc) --shuffle t[0-9]*.sh
```

> [!NOTE] You can also do this with `make test` or use any testing harness which can speak TAP. `prove` can run concurrently. `-j$(nproc)` runs tests using all available CPUs in parallel, but the job count can be adjusted as needed. `shuffle` randomizes the order the tests are run in, which makes them resilient against unwanted inter-test dependencies. `prove` also makes the output nicer.

Go ahead and commit this change, as well.

Getting Ready to Share: Anatomy of a Patch Series

You may have noticed already that the Git project performs its code reviews via emailed patches, which are then applied by the maintainer when they are ready and approved by the community. The Git project does not accept contributions from pull requests, and the patches emailed for review need to be formatted a specific way.

Before taking a look at how to convert your commits into emailed patches, let's analyze what the end result, a "patch series", looks like. Here is an https://lore.kernel.org/git/pull.1218.git.git.1645209647.gitgitgadget@gmail.com/[example] of the summary view for a patch series on the web interface of the https://lore.kernel.org/git/[Git mailing list archive]:

```
2022-02-18 18:40 [PATCH 0/3] libify reflog John Cai via GitGitGadget
2022-02-18 18:40 ` [PATCH 1/3] reflog: libify delete reflog function and helpers John Cai via
GitGitGadget
2022-02-18 19:10   ` Ævar Arnfjörð Bjarmason [this message]
2022-02-18 19:39     ` Taylor Blau
2022-02-18 19:48       ` Ævar Arnfjörð Bjarmason
```

```
2022-02-18 19:35   ` Taylor Blau
2022-02-21  1:43     ` John Cai
2022-02-21  1:50       ` Taylor Blau
2022-02-23 19:50         ` John Cai
2022-02-18 20:00   ` // other replies elided
2022-02-18 18:40 ` [PATCH 2/3] reflog: call reflog_delete from reflog.c John Cai via
GitGitGadget
2022-02-18 19:15   ` Ævar Arnfjörð Bjarmason
2022-02-18 20:26     ` Junio C Hamano
2022-02-18 18:40 ` [PATCH 3/3] stash: call reflog_delete from reflog.c John Cai via
GitGitGadget
2022-02-18 19:20   ` Ævar Arnfjörð Bjarmason
2022-02-19  0:21     ` Taylor Blau
2022-02-22  2:36     ` John Cai
2022-02-22 10:51       ` Ævar Arnfjörð Bjarmason
2022-02-18 19:29 ` [PATCH 0/3] libify reflog Ævar Arnfjörð Bjarmason
2022-02-22 18:30 ` [PATCH v2 0/3] libify reflog John Cai via GitGitGadget
2022-02-22 18:30   ` [PATCH v2 1/3] stash: add test to ensure reflog --rewrite --updatref
behavior John Cai via GitGitGadget
2022-02-23  8:54     ` Ævar Arnfjörð Bjarmason
2022-02-23 21:27       ` Junio C Hamano
// continued
```

We can note a few things:

- Each commit is sent as a separate email, with the commit message title as subject, prefixed with "[PATCH *i*/*n*]" for the *i*-th commit of an *n*-commit series.
- Each patch is sent as a reply to an introductory email called the *cover letter* of the series, prefixed "[PATCH 0/*n*]".
- Subsequent iterations of the patch series are labelled "PATCH v2", "PATCH v3", etc. in place of "PATCH". For example, "[PATCH v2 1/3]" would be the first of three patches in the second iteration. Each iteration is sent with a new cover letter (like "[PATCH v2 0/3]" above), itself a reply to the cover letter of the first iteration (more on that below).

> [!NOTE] A single-patch topic is sent with "[PATCH]", "[PATCH v2]", etc. without *i*/*n* numbering (in the above thread overview, no single-patch topic appears, though).

The cover letter

In addition to an email per patch, the Git community also expects your patches to come with a cover letter. This is an important component of change submission as it explains to the community from a high level what you're trying to do, and why, in a way that's more apparent than just looking at your patches.

The title of your cover letter should be something which succinctly covers the purpose of your entire topic branch. It's often in the imperative mood, just like our commit message titles. Here is how we'll title our series:

Add the 'psuh' command ---

The body of the cover letter is used to give additional context to reviewers. Be sure to explain anything your patches don't make clear on their own, but remember that since the cover letter is not recorded in the commit history, anything that might be useful to future readers of the repository's history should also be in your commit messages.

Here's an example body for `psuh`:

```
Our internal metrics indicate widespread interest in the command
git-psuh - that is, many users are trying to use it, but finding it is
unavailable, using some unknown workaround instead.

The following handful of patches add the psuh command and implement some
handy features on top of it.

This patchset is part of the MyFirstContribution tutorial and should not
be merged.
```

At this point the tutorial diverges, in order to demonstrate three different methods of formatting your patchset and getting it reviewed.

The first method to be covered is GitGitGadget, which is useful for those already familiar with GitHub's common pull request workflow. This method requires a GitHub account.

The second method to be covered is `git send-email`, which can give slightly more fine-grained control over the emails to be sent. This method requires some setup which can change depending on your system and will not be covered in this tutorial.

The third method to be covered is `b4`, which builds on top of `git format-patch` and `git send-email`. This method is the recommended way to submit patches via mail as it automates a lot of the bookkeeping required by `git send-email`.

Regardless of which method you choose, your engagement with reviewers will be the same; the review process will be covered after the sections on GitGitGadget, `git` `send-email` and `b4`.

Sending Patches via GitGitGadget

One option for sending patches is to follow a typical pull request workflow and send your patches out via GitGitGadget. GitGitGadget is a tool created by Johannes Schindelin to make life as a Git contributor easier for those used to the GitHub PR workflow. It allows contributors to open pull requests against its mirror of the Git project, and does some magic to turn the PR into a set of emails and send them out for you. It also runs the Git continuous integration suite for you. It's documented at https://gitgitgadget.github.io/.

Forking `git/git` on GitHub

Before you can send your patch off to be reviewed using GitGitGadget, you will need to fork the Git project and upload your changes. First thing - make sure you have a GitHub account.

Head to the GitHub mirror and look for the Fork button. Place your fork wherever you deem appropriate and create it.

Uploading to Your Own Fork

To upload your branch to your own fork, you'll need to add the new fork as a remote. You can use `git` `remote` `-v` to show the remotes you have added already. From your new fork's page on GitHub, you can press "Clone or download" to get the URL; then you need to run the following to add, replacing your own URL and remote name for the examples provided:

```
$ git remote add remotename git@github.com:remotename/git.git
```

or to use the HTTPS URL:

```
$ git remote add remotename https://github.com/remotename/git/.git
```

Run `git remote -v` again and you should see the new remote showing up. `git fetch remotename` (with the real name of your remote replaced) in order to get ready to push.

Next, double-check that you've been doing all your development in a new branch by running `git branch`. If you didn't, now is a good time to move your new commits to their own branch.

As mentioned briefly at the beginning of this document, we are basing our work on `master`, so go ahead and update as shown below, or using your preferred workflow.

```
$ git checkout master
$ git pull -r
$ git rebase master psuh
```

Finally, you're ready to push your new topic branch! (Due to our branch and command name choices, be careful when you type the command below.)

```
$ git push remotename psuh
```

Now you should be able to go and check out your newly created branch on GitHub.

Sending a PR to GitGitGadget

In order to have your code tested and formatted for review, you need to start by opening a Pull Request against either `gitgitgadget/git` or `git/git`. Head to https://github.com/gitgitgadget/git or https://github.com/git/git and open a PR either with the "New pull request" button or the convenient "Compare & pull request" button that may appear with the name of your newly pushed branch.

The differences between using `gitgitgadget/git` and `git/git` as your base can be found [here](https://gitgitgadget.github.io/#should-i-use-gitgitgadget-on-gitgitgadgets-git-fork-or-on-gits-github-mirror)

Review the PR's title and description, as they're used by GitGitGadget respectively as the subject and body of the cover letter for your change. Refer to "The cover letter" above for advice on how to title your submission and what content to include in the description.

> [!NOTE] For single-patch contributions, your commit message should already be meaningful and explain at a high level the purpose (what is happening and why) of your patch, so you usually do not need any additional context. In that case,

> remove the PR description that GitHub automatically generates from your commit message (your PR description should be empty). If you do need to supply even more context, you can do so in that space and it will be appended to the email that GitGitGadget will send, between the three-dash line and the diffstat (see Bonus Chapter: One-Patch Changes for how this looks once submitted).

When you're happy, submit your pull request.

Running CI and Getting Ready to Send

If it's your first time using GitGitGadget (which is likely, as you're using this tutorial) then someone will need to give you permission to use the tool. As mentioned in the GitGitGadget documentation, you just need someone who already uses it to comment on your PR with `/allow` `<username>`. GitGitGadget will automatically run your PRs through the CI even without the permission given but you will not be able to `/submit` your changes until someone allows you to use the tool.

> [!NOTE] You can typically find someone who can `/allow` you on GitGitGadget by either examining recent pull requests where someone has been granted `/allow` (Search: is:pr is:open "/allow"), in which case both the author and the person who granted the `/allow` can now `/allow` you, or by inquiring on the #git-devel IRC channel on Libera Chat linking your pull request and asking for someone to `/allow` you.

If the CI fails, you can update your changes with `git` `rebase` `-i` and push your branch again:

```
$ git push -f remotename psuh
```

In fact, you should continue to make changes this way up until the point when your patch is accepted into `next`.

Sending Your Patches

Now that your CI is passing and someone has granted you permission to use GitGitGadget with the `/allow` command, sending out for review is as simple as commenting on your PR with `/submit`.

Updating With Comments

Skip ahead to Responding to Reviews for information on how to reply to review comments you will receive on the mailing list.

Once you have your branch again in the shape you want following all review comments, you can submit again:

```
$ git push -f remotename psuh
```

Next, go look at your pull request against GitGitGadget; you should see the CI has been kicked off again. Now while the CI is running is a good time for you to modify your description at the top of the pull request thread; it will be used again as the cover letter. You should use this space to describe what has changed since your previous version, so that your reviewers have some idea of what they're looking at. When the CI is done running, you can comment once more with `/submit` - GitGitGadget will automatically add a v2 mark to your changes.

Sending Patches with `git send-email`

If you don't want to use GitGitGadget, you can also use Git itself to mail your patches. Some benefits of using Git this way include finer grained control of subject line (for example, being able to use the tag [RFC PATCH] in the subject) and being able to send a ``dry run'' mail to yourself to ensure it all looks good before going out to the list.

Prerequisite: Setting Up `git send-email`

Configuration for `send-email` can vary based on your operating system and email provider, and so will not be covered in this tutorial, beyond stating that in many distributions of Linux, `git-send-email` is not packaged alongside the typical `git` install. You may need to install this additional

package; there are a number of resources online to help you do so. You will also need to determine the right way to configure it to use your SMTP server; again, as this configuration can change significantly based on your system and email setup, it is out of scope for the context of this tutorial.

Preparing Initial Patchset

Sending emails with Git is a two-part process; before you can prepare the emails themselves, you'll need to prepare the patches. Luckily, this is pretty simple:

```
$ git format-patch --cover-letter -o psuh/ --base=auto psuh@{u}..psuh
```

1. The `--cover-letter` option tells `format-patch` to create a cover letter template for you. You will need to fill in the template before you're ready to send - but for now, the template will be next to your other patches.

1. The `-o` `psuh/` option tells `format-patch` to place the patch files into a directory. This is useful because `git` `send-email` can take a directory and send out all the patches from there.

1. The `--base=auto` option tells the command to record the "base commit", on which the recipient is expected to apply the patch series. The `auto` value will cause `format-patch` to compute the base commit automatically, which is the merge base of tip commit of the remote-tracking branch and the specified revision range.

1. The `psuh@{u}..psuh` option tells `format-patch` to generate patches for the commits you created on the `psuh` branch since it forked from its upstream (which is `origin/master` if you followed the example in the "Set up your workspace" section). If you are already on the `psuh` branch, you can just say `@{u}`, which means "commits on the current branch since it forked from its upstream", which is the same thing.

The command will make one patch file per commit. After you run, you can go have a look at each of the patches with your favorite text editor and make sure everything looks alright; however, it's not recommended to make code fixups via the patch file. It's a better idea to make the change the normal way using `git` `rebase` `-i` or by adding a new commit than by modifying a patch.

> [!NOTE] Optionally, you can also use the `--rfc` flag to prefix your patch subject with `[RFC` `PATCH]''` `instead` `of` [PATCH]''. RFC stands for `request` `for` `comments''` `and` `indicates` `that` `while` `your` `code` `isn't` `quite` `ready` `for` `submission,` `you'd` `like` `to` `begin` `the` `code` `review` `process.` `This` `can` `also` `be` `used` `when` `your` `patch` `is` `a` `proposal,` `but` `you` `aren't` `sure` `whether` `the` `community` `wants` `to` `solve` `the` `problem` `with` `that` `approach` `or` `not` `-` `to` `conduct` `a` `sort` `of` `design` `review.` `You` `may` `also` `see` `on` `the` `list`

> `patches` `marked` WIP'' - this means they are incomplete but want reviewers to look at what they have so far. You can add this flag with `--subject-prefix=WIP`.

Check and make sure that your patches and cover letter template exist in the directory you specified - you're nearly ready to send out your review!

Preparing Email

Since you invoked `format-patch` with `--cover-letter`, you've already got a cover letter template ready. Open it up in your favorite editor.

You should see a number of headers present already. Check that your `From:` header is correct. Then modify your `Subject:` (see above for how to choose good title for your patch series):

```
Subject: [PATCH 0/7] Add the 'psuh' command
```

Make sure you retain the ``[PATCH 0/X]'' part; that's what indicates to the Git community that this email is the beginning of a patch series, and many reviewers filter their email for this type of flag.

You'll need to add some extra parameters when you invoke `git` `send-email` to add the cover letter.

Next you'll have to fill out the body of your cover letter. Again, see above for what content to include.

The template created by `git` `format-patch` `--cover-letter` includes a diffstat. This gives reviewers a summary of what they're in for when reviewing your topic. The one generated for `psuh` from the sample implementation looks like this:

```
Documentation/git-psuh.adoc | 40 ++++++++++++++++++++
Makefile                    |  1 +
builtin.h                   |  1 +
builtin/psuh.c              | 73 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
git.c                       |  1 +
t/t9999-psuh-tutorial.sh    | 12 +++++++
6 files changed, 128 insertions(+)
create mode 100644 Documentation/git-psuh.adoc
create mode 100644 builtin/psuh.c
create mode 100755 t/t9999-psuh-tutorial.sh
```

Finally, the letter will include the version of Git used to generate the patches. You can leave that string alone.

Sending Email

At this point you should have a directory `psuh/` which is filled with your patches and a cover letter. Time to mail it out! You can send it like this:

```
$ git send-email --to=target@example.com psuh/*.patch
```

> [!NOTE] Check `git` `help` `send-email` for some other options which you may find valuable, such as changing the Reply-to address or adding more CC and BCC lines.

not part of the core `git` binary and must be called directly. Clone the Git codebase and run `perl` `contrib/contacts/git-contacts`.]

> [!NOTE] If you’re not sure whom to CC, running `contrib/contacts/git-contacts` can list potential reviewers. In addition, you can do `git` `send-email` `--cc-cmd=’perl` `contrib/contacts/git-contacts’` `feature/*.patch`footnoteref:[contrib-scripts,Scripts under `contrib/` are + to automatically pass this list of emails to `send-email`.

> [!NOTE] When you are sending a real patch, it will go to git@vger.kernel.org - but please don’t send your patchset from the tutorial to the real mailing list! For now, you can send it to yourself, to make sure you understand how it will look.

> [!NOTE] After sending your patches, you can confirm that they reached the mailing list by visiting https://lore.kernel.org/git/. Use the search bar to find your name or the subject of your patch. If it appears, your email was successfully delivered.

After you run the command above, you will be presented with an interactive prompt for each patch that’s about to go out. This gives you one last chance to edit or quit sending something (but again, don’t edit code this way). Once you press `y` or `a` at these prompts your emails will be sent!

Congratulations!

Awesome, now the community will drop everything and review your changes. (Just kidding - be patient!)

Sending v2

This section will focus on how to send a v2 of your patchset. To learn what should go into v2, skip ahead to Responding to Reviews for information on how to handle comments from reviewers.

We'll reuse our `psuh` topic branch for v2. Before we make any changes, we'll mark the tip of our v1 branch for easy reference:

```
$ git checkout psuh
$ git branch psuh-v1
```

Refine your patch series by using `git` `rebase` `-i` to adjust commits based upon reviewer comments. Once the patch series is ready for submission, generate your patches again, but with some new flags:

```
$ git format-patch -v2 --cover-letter -o psuh/ --range-diff master..psuh-v1 master..
```

The `--range-diff` `master..psuh-v1` parameter tells `format-patch` to include a range-diff between `psuh-v1` and `psuh` in the cover letter (see linkgit:git-range-diff[1]). This helps tell reviewers about the differences between your v1 and v2 patches.

The `-v2` parameter tells `format-patch` to output your patches as version "2". For instance, you may notice that your v2 patches are all named like `v2-000n-my-commit-subject.patch`. `-v2` will also format your patches by prefixing them with "[PATCH v2]" instead of "[PATCH]", and your range-diff will be prefaced with "Range-diff against v1".

After you run this command, `format-patch` will output the patches to the `psuh/` directory, alongside the v1 patches. Using a single directory makes it easy to refer to the old v1 patches while proofreading the v2 patches, but you will need to be careful to send out only the v2 patches. We will use a pattern like `psuh/v2-*.patch` (not `psuh/*.patch`, which would match v1 and v2 patches).

Edit your cover letter again. Now is a good time to mention what's different between your last version and now, if it's something significant. You do not need the exact same body in your second cover letter; focus on explaining to reviewers the changes you've made that may not be as visible.

You will also need to go and find the Message-ID of your first cover letter. You can either note it when you send the first series, from the output of `git` `send-email`, or you can look it up on the mailing list. Find your cover letter in the archives, click on it, then click "permalink" or "raw" to

reveal the Message-ID header. It should match:

```
Message-ID: <foo.12345.author@example.com>
```

Your Message-ID is `<` `foo.12345.author@example.com` `>`. This example will be used below as well; make sure to replace it with the correct Message-ID for your **first cover letter** - that is, for any subsequent version that you send, always use the Message-ID from v1.

While you're looking at the email, you should also note who is CC'd, as it's common practice in the mailing list to keep all CCs on a thread. You can add these CC lines directly to your cover letter with a line like so in the header (before the Subject line):

```
CC: author@example.com, Othe R <other@example.com>
```

Now send the emails again, paying close attention to which messages you pass in to the command:

```
$ git send-email --to=target@example.com
         --in-reply-to="<foo.12345.author@example.com>"
         psuh/v2-*.patch
```

Bonus Chapter: One-Patch Changes

In some cases, your very small change may consist of only one patch. When that happens, you only need to send one email. Your commit message should already be meaningful and explain at a high level the purpose (what is happening and why) of your patch, but if you need to supply even more context, you can do so below the `---` in your patch. Take the example below, which was generated with `git` `format-patch` on a single commit, and then edited to add the content between the `---` and the diffstat.

```
From 1345bbb3f7ac74abde040c12e737204689a72723 Mon Sep 17 00:00:00 2001
From: A U Thor <author@example.com>
Date: Thu, 18 Apr 2019 15:11:02 -0700
Subject: [PATCH] README: change the grammar

I think it looks better this way. This part of the commit message will
end up in the commit-log.

Signed-off-by: A U Thor <author@example.com>
```

```
---
Let's have a wild discussion about grammar on the mailing list. This
part of my email will never end up in the commit log. Here is where I
can add additional context to the mailing list about my intent, outside
of the context of the commit log. This section was added after `git
format-patch` was run, by editing the patch file in a text editor.

 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

diff --git a/README.md b/README.md
index 88f126184c..38da593a60 100644
--- a/README.md
+++ b/README.md
@@ -3,7 +3,7 @@
 Git - fast, scalable, distributed revision control system
 =========================================================

-Git is a fast, scalable, distributed revision control system with an
+Git is a fast, scalable, and distributed revision control system with an
 unusually rich command set that provides both high-level operations
 and full access to internals.

--
2.21.0.392.gf8f6787159e-goog
```

Sending Patches with `b4`

`b4` is a tool that builds on top of `git format-patch` and `git send-email`. It automates much of the bookkeeping involved in sending a patch series to a mailing-list-based project.

Refer to the b4 documentation for a full reference.

Preparing a Patch Series

`b4` tracks your patch series as a branch. To start tracking the `psuh` branch you have been working on, run:

```
$ b4 prep --enroll master
```

This enrolls the current branch, using `master` as the base of the topic. `b4` manages the cover letter as part of the branch, so you can edit it at any time with:

```
$ b4 prep --edit-cover
```

The cover letter not only tracks the content of the top-level mail, but also the set of recipients. You can add recipients by adding `To:` and `Cc:` trailer lines.

Sending the Patches

Before sending the series out for real, you can inspect what `b4` would send by passing `--dry-run`:

```
$ b4 send --dry-run
```

Once you are happy with the result, send the series with:

```
$ b4 send
```

Sending v2

When you are ready to send a new iteration of your series, refine your patches as usual using linkgit:git-rebase[1]. Note that you typically want to rebase on top of the cover letter. You can configure an alias to enable easy rebases going forward:

$ git config set alias.b4-rebase 'rebase "HEAD^{/--- b4-submit-tracking ---}"' $ git b4-rebase -i ---

Before sending out the new version you should also update the cover letter with `b4` `prep` `--edit-cover` to note the relevant changes compared to the previous version. You can inspect the changes between the two versions with `b4` `prep` `--compare-to=v1`.

Same as with the first version, you can use `b4` `send` to send out the second version. `b4` automatically bumps the version to `v2`, generates the range-diff against the previous iteration, and threads the new series as a reply to the cover letter of the first version.

Configure b4

`b4` can be configured via linkgit:git-config[1]. In addition to that, projects can have their own set of defaults in `.b4-config` in the root tree, which also uses Git's config format. The user's configuration always takes precedence over the per-project defaults.

Refer to the b4 config documentation for more information on the available options.

My Patch Got Emailed - Now What?

Please give reviewers enough time to process your initial patch before sending an updated version. That is, resist the temptation to send a new version immediately, because others may have already started reviewing your initial version.

While waiting for review comments, you may find mistakes in your initial patch, or perhaps realize a different and better way to achieve the goal of the patch. In this case you may communicate your findings to other reviewers as follows:

- If the mistakes you found are minor, send a reply to your patch as if you were a reviewer and mention that you will fix them in an updated version.

- On the other hand, if you think you want to change the course so drastically that reviews on the initial patch would be a waste of time (for everyone involved), retract the patch immediately with a reply like "I am working on a much better approach, so please ignore this patch and wait for the updated version."

Now, the above is a good practice if you sent your initial patch prematurely without polish. But a better approach of course is to avoid sending your patch prematurely in the first place.

Please be considerate of the time needed by reviewers to examine each new version of your patch. Rather than seeing the initial version right now (followed by several "oops, I like this version better than the previous one" patches over 2 days), reviewers would strongly prefer if a single polished version came 2 days later instead, and that version with fewer mistakes were the only one they would need to review.

This consideration applies not only when going from the initial patch to v2, but also to later iterations of the same series. There is no fixed rule for how long to wait before sending a new version. A useful default is to send at most one new version of the same patch series per day. This gives multiple reviewers time to comment, gives reviewers across time zones a fair chance to participate, lets you batch feedback together, and gives you time to think through the comments you received. Knowing that you should not immediately send another version also encourages you to review the patches more carefully before sending them, catch small mistakes such as typos and

off-by-one errors yourself, and let reviewers spend more of their attention on design, algorithms, and other substantial issues.

The right timing depends on the topic and the feedback. Larger series usually need more review time. If the only comments so far are minor, such as typo fixes, it often makes sense to wait a little longer in case deeper reviews are still coming. If the comments call for substantial rework, do not rush out an updated version before you have reviewed the larger changes carefully. Instead, reply to the review that prompted the rewrite, say that you are preparing a substantial rework, and mention which parts of the current series will become obsolete so reviewers can avoid spending time on them until the updated series is ready.

Responding to Reviews

After a few days, you will hopefully receive a reply to your patchset with some comments. Woohoo! Now you can get back to work.

It’s good manners to reply to each comment in the mailing list discussion instead of letting the next version of your patch be your only response. Tell the reviewer whether you plan to make the suggested change, keep the original, or pursue a different approach. This way reviewers can respond to your reasoning before you spend time preparing a version they may not agree with, and later do not need to inspect your v2 to figure out whether you implemented their comment or not.

Reviewers may ask you about what you wrote in the patchset, either in the proposed commit log message or in the changes themselves. You should answer these questions in your response messages, but often the reason why reviewers asked these questions to understand what you meant to write is because your patchset needed clarification to be understood.

Do not be satisfied by just answering their questions in your response and hear them say that they now understand what you wanted to say. Update your patches to clarify the points reviewers had trouble with, and prepare your v2; the words you used to explain your v1 to answer reviewers’ questions may be useful thing to use. Your goal is to make your v2 clear enough so that it becomes unnecessary for you to give the same explanation to the next person who reads it.

If you are going to push back on a comment, be polite and explain why you feel your original is better; be prepared that the reviewer may still disagree with you, and the rest of the community may weigh in on one side or the other. As with all code reviews, it’s important to keep an open mind to doing something a different way than you originally planned; other reviewers have a different perspective on the project than you do, and may be thinking of a valid side effect which had not occurred to you. It is always okay to ask for clarification if you aren’t sure why a change was suggested, or what the reviewer is asking you to do.

When replying to review comments, quote only the parts of the message that are relevant to your response. It is usually helpful to trim away unrelated context, such as large portions of the patch that are not being discussed, while keeping enough quoted text for readers to understand what you

are responding to.

Make sure your email client has a plaintext email mode and it is turned on; the Git list rejects HTML email. Please also follow the mailing list etiquette outlined in the Maintainer's Note, which are similar to etiquette rules in most open source communities surrounding bottom-posting and inline replies.

When you're making changes to your code, it is cleanest - that is, the resulting commits are easiest to look at - if you use `git rebase -i` (interactive rebase). Take a look at this overview from O'Reilly. The general idea is to modify each commit which requires changes; this way, instead of having a patch A with a mistake, a patch B which was fine and required no upstream reviews in v1, and a patch C which fixes patch A for v2, you can just ship a v2 with a correct patch A and correct patch B. This is changing history, but since it's local history which you haven't shared with anyone, that is okay for now! (Later, it may not make sense to do this; take a look at the section below this one for some context.)

After Review Approval

The Git project has four integration branches: `seen`, `next`, `master`, and `maint`. Your change will be placed into `seen` fairly early on by the maintainer while it is still in the review process; from there, when it is ready for wider testing, it will be merged into `next`. Plenty of early testers use `next` and may report issues. Eventually, changes in `next` will make it to `master`, which is typically considered stable. Finally, when a new release is cut, `maint` is used to base bugfixes onto. As mentioned at the beginning of this document, you can read `Documents/SubmittingPatches` for some more info about the use of the various integration branches.

Back to now: your code has been lauded by the upstream reviewers. It is perfect. It is ready to be accepted. You don't need to do anything else; the maintainer will merge your topic branch to `next` and life is good.

However, if you discover it isn't so perfect after this point, you may need to take some special steps depending on where you are in the process.

If the maintainer has announced in the "What's cooking in git.git" email that your topic is marked for `next` - that is, that they plan to merge it to `next` but have not yet done so - you should send an email asking the maintainer to wait a little longer: "I've sent v4 of my series and you marked it for `next`, but I need to change this and that - please wait for v5 before you merge it."

If the topic has already been merged to `next`, rather than modifying your patches with `git rebase -i`, you should make further changes incrementally - that is, with another commit, based on top of the maintainer's topic branch as detailed in https://github.com/gitster/git. Your work is still in the same topic but is now incremental, rather than a wholesale rewrite of the topic branch.

The topic branches in the maintainer’s GitHub are mirrored in GitGitGadget, so if you’re sending your reviews out that way, you should be sure to open your PR against the appropriate GitGitGadget/Git branch.

If you’re using `git send-email`, you can use it the same way as before, but you should generate your diffs from `<topic>..<mybranch>` and base your work on `<topic>` instead of `master`.

□□□ #2
□ □□□ □□□ 2026-10-31 19:44:55 UTC
□ □□□ □□□ 2026-10-31 19:44:57 UTC