Git for Teams of One or More

Emma Jane Hogbin Westby

Contents

Introduction

Welcome and Set-up

I've been teaching version control for almost as long as I've been using it. Over the years I've refined how I've taught this topic, and have learned a lot about what doesn't work in the traditional ways we currently teach version control. In this learning series, I'm going to be doing things a little bit differently. Instead of focusing on the commands you'll be running, I'm going to try and start with WHY you'd want to do something, and then show you HOW you'd go about implementing the solution.

As a developer, I've learned a lot of little hacks to help me be more efficiently. I'm still not fantastic at the command line. I haven't replaced myself with a tiny shell script, but I am very comfortable working from the command line. I enjoy the elegance of not having to click eleventy buttons on eight screens, and instead just issuing a few commands. I've also found it's a lot easier to write cheat sheets for the command line than for a GUI. And there seem to be a lot more answers on Stack Overflow, but that could just be Google optimizing the answers for the types of things I'm normally looking for.

As this learning series is a "watch over my shoulder" type of series, we're going to work in my usual fashion: from the command line. Unlike watching over my actual shoulder, you'll be able to pause the video, and loop back over anything I've said if you missed it the first time.

Throughout this learning series I'm going to assume you have a number of things setup and ready to go. I'll be working from the command line. If you prefer to use a GUI, you'll need to figure out how these commands map onto your software application. I encourage you to try the command line with me.

One of the reasons why people like having a GUI is that it helps them to create a visual map of what they're working on. Instead of relying on software to provide this map, I'll be using diagrams. I encourage you to grab a drawing tool (paper and pen work great!) and sketch along with me. Creating your own diagrams will help the learning process as the movement helps to build yet another neural pathway to the information.

Lesson Objectives

By the end of this lesson, you will be prepared for learning Git.

Self-Check

I have assembled the following:

- Drawing tools for diagrams (pencil/pen and paper is fine!).
- Git is installed at the command line.

Lesson Summary

Working at the command line is universal. It may require some additional configuration for Windows, but it allows us to share commands across operating systems without having to worry about the discrepancy between GUIs.

When learning new topics, creating a drawing to represent information will help you to create the neural pathways which make it easier to recall the information later on.

Warm-up Exercise

Being able to describe in words, or with pictures, how something works, converts it from the abstract, into something which can be programmed. Any time we are able to use words in a linear fashion (i.e. talking through a problem), we are able to find the corresponding commands that we'll need to issue to make our process work. If you are not able to diagram, or describe your problem, it will be near impossible to find the matching commands you need to run.

Sketch the assembly line for your code as it exists currently, or as you think you would like it to work. Examples: centralized, peer review, automated gate keeper.

Lesson Objectives

By the end of this lesson, you should be able to sketch the relationship between the people on your team, and the tasks they perform.

Self-Check

Does your diagram summarize how your team currently works together (or how you'd like them to work together)?

Summary

There are three fairly common workflows for teams:

- centralized: code checked into a single, central server
- peer review: a person checks the code before merging it to a main development branch
- automated gate keeper: a test suite checks the code before merging it into a main development branch

These examples aren't exhaustive, and they aren't mutually exclusive. Your workflow may incorporate bits and pieces of these, or use something different. We'll work on refining your understanding of how these diagrams relate to git commands throughout the rest of this learning series.

Getting Started with Hosted Git Repositories

Most of the people I've worked with are starting out with a project they need to collaborate with. They aren't starting with an empty folder, but rather with an existing project. I personally find it easier to tinker with something rather than stare at a blank screen. So the first thing we're going to do is set you up with a practice repository that I've created. You won't be able to edit *my* copy of the project, but you will be able to make edits to your own copy.

Overview of permission strategies

Let's start with a review of permission strategies and the way a project can be setup with Git.

Overview of permission strategies:

- *Centralized.* Commonly seen for centralized systems, such as Subversion. In this system, everyone is committing into the same repository on a single machine. When you're working with a distributed version control system, such as Git, you're actually using a centralized model. This model often has finer grained access on who can make a commit, and where they can commit to.
- *Patched.* The first step away from a centralized system, is to work in patch files. In this case, there is still a central, canonical place where code is stored, like in the centralized system. But in this case, very few people are allowed to commit to the central repository. Instead, code changes are shared as patch files. This permission strategy is lesson common today, but it's still used by the Linux kernel team, and by Drupal.
- *Forked.* Most open source projects will use a forking permission strategy. In this case, we are finally seeing the difference (and power) of a distributed version control system. In a forking workflow, there are multiple copies of the repository. I have my copy; you have your copy. And through a social convention, we decide which copy is the canonical (or primary) source. This permission strategy is used to limit who is allowed to make commits back into the canonical source for a project. In other words: it is a limiting strategy.
- *Branched.* Whereas a forking strategy gives very limited access to a small number of people, a branching strategy is a permissive strategy. The branching strategy is typically used by internal teams (and the teams responsible for the canonical repository of a bigger project). In this strategy it is only by convention that people choose which branches they want to save their commits to. There are no restrictions once you have access.

Permission strategies and branching strategies are often conflated in articles about version control. This can make it confusing. When examining a diagram, take a look at the symbols used. If there are dots representing individual commits, you are probably looking at a *branching strategy* diagram. We'll get to those a bit later in the series.

Lesson Objectives

By the end of this lesson, you will be able to identify four different permission strategies used for version control, and explain in each strategy, who may make commits to a repository.

Self-Check

Identify the reasons why the identified strategies was selected:

- Large open source project may use a forking workflow.
- An internal team project uses a branching workflow.

Lesson Summary

There are four main permission strategies used with distributed version control systems:

- Centralized is how you work at your own work station. Changes are saved into your local repoitory, unlike in a centralized *system* where the changes were saved into a remote repository.
- Patching strategy is where each person has a local repository, and shares their work with others by sending patch files, which identify how to "patch up" a repository with a common ancestor so it looks like the work the developer as done in their own copy of the repository.
- Forked strategy is where each person creates a copy of the canonical repository, and maintains a connection to it. Proposed changes are made by submitting a "pull request" or "merge request" to the upstream project.
- Branching strategy is where every person on the team has write-access to the project. Instead of waiting for a pull request (or merge request) to be accepted, they are able to save their changes directly to the repository, without additional permission from another person.

Creating a GitLab Account

GitLab is a free code hosting platform which allows you to create private, and public repositories. It's also an open source product you can install behind your own firewall if you're working with a large team. It's not the most popular system out there, but I believe it offers the most flexibility to the widest range of people watching these videos. We'll get started with GitLab, and later in the lessons, I'll show you how to migrate your project to GitHub, which is currently one of the most popular code hosting platforms for open source projects. Git is distributed because work can be easily shared across multiple hosting systems. Create a GitLab account to store your first remote repository.

Lesson Objectives

By the end of this lesson, you will be able to create an account on GitLab.

Self-Check

Ensure you have a GitLab account before proceding to the next lesson.

Summary

1. Navigate to www.gitlab.com.

2. At the bottom of the screen locate the link to create an account.
3. Complete the on-screen instructions to create your account.

Gotchas

GitLab releases new software every month. If the interface is significantly different, check the instructions on GitForTeams.com. I have made a backup of GitLab which uses same interface as these video lessons.

Adding Your SSH Keys

SSH keys are a way to identify trusted computers, without using passwords. When you are using code hosting systems, these SSH keys will allow you to effectively "log in" to the system, and either retrieve, or save, code as if you had logged into to your account.

Lesson Objectives

By the end of this lesson, you will be able to locate, and add your SSH keys to GitLab. If do not have SSH keys already, you will also be able to create a new SSH key pair.

Self-Check

Your SSH public key has been uploaded to GitLab.

Summary

Locate, or Generate SSH Keys:

1. At the command line, run: `cat ~/.ssh/id_rsa.pub`. If there is a blob of text printed to the screen beginning with `ssh-rsa` or `ssh-dsa`, skip the next step.
2. To generate new keys, run: `ssh-keygen -t rsa -C "$your_email"` (replace with your actual email)
3. To display your new public key, run: `cat ~/.ssh/id_rsa.pub`.
4. Copy the text for your public key.

Add Your Public Key to GitLab:

1. Log in to GitLab.

2. Navigate to your account (top right).
3. Locate and click on the tab for SSH keys.
4. Click "Add SSH Keys".
5. In the form, paste your SSH public key. (The title is optional.)
6. Click save.

Forking Your First Project

We are forking so that you can upload your changes to your own repository. If you cloned from this project directly to your computer you wouldn't be able to upload your changes to the project (only "privileged" users can do that).

Lesson Objectives

By the end of this lesson, you will be able to create a fork of a remote repository.

Self-Check

In your GitLab account, you should now have a new project with its upstream set to `gitforteams/workshop`.

Summary

Fork the GitLab sample project:

1. Navigate to the project page: https://gitlab.com/gitforteams/workshop
2. Locate and click on the button labeled "fork". https://gitlab.com/gitforteams/workshop/fork

Tada.

Privatizing Your Repository

Adjust the visibility of your forked repository to PRIVATE. Although this is completely optional, a lot of people feel more comfortable making mistakes when they know no one will see them.

Private repositories are also useful if you want to use GitLab for private projects, such as client work, and don't want others to be able to see the code you are creating.

Lesson Objectives

By the end of this lesson, you will be able to alter the permission settings of your repository so that no one else can view its contents.

Self-Check

When you log out of GitLab, are you still able to navigate to your project page and see your work?

Summary

Adjust the settings of your forked repository so that the project is private.

1. Navigate to your account page (click the avatar, top right).
2. From the right sidebar, select the project you've just forked.
3. Click the "settings" tab.
4. Scroll down and change the visibility settings to "Private".
5. Scroll to the bottom of the page and click "save change".

Downloading a Remote Repository

Overview of Git at the Command Line

Open up the command line, and check to see which version of Git you have installed. Look at Git's short help, list of all commands, and a single command. Recommend Stack Overflow / Google in additional to the command line help.

Lesson Objectives

By the end of this lesson, you will be able to retrieve help for git's commands from the command line.

Self-Check

You are able to check which version of git is installed from the command line.

Summary

The commands we used in this lesson allowed us to read the manual pages for git from the command line.

- `git`
- `git help` (same as running git without a parameter)
- `git help clone`

For additional information on the specifics of using any one command, do a Google search. You'll probably end up at Stack Overflow. I've also put together a comprehensive list of resources for learning git.

Cloning your GitLab repository

With your fork of a projec setup, it's now time to download the project so that you can work on it locally. There isn't actually a fork command in git! When you create a fork of a project, you are merely cloning, or copying, the repository to a new location. Once you have a local clone of your repository, you will have essentially setup a chain between the three repositories. The original repository (my copy) is the "upstream project". It might receive contributions from any number of people, but they will always need to be approved by me. The next repository in your chain, is the repository which you forked from my project. This one you have write access to. Perhaps other people will be interested in the adaptations you've been making to your copy, and will be interested in helping you with yours (this is where the concept of "fork" comes in, as the two repositories may diverge over time, just like the tines on a fork). The third, and final repository in the chain, is your local repository. From this repository you have one connection to the remote repository you cloned from. You could add others as well. In subsequent lessons we'll talk about why you might want to do this.

Cloning a project is very straight forward. You will need to:

1. Navigate to the project page for the repository you want to clone.
2. Locate the instructions on how to clone, copy the instructions.
3. At the command line, navigate to the directory where you want to download the repository to.
4. Paste the command line instructions.

Generally these instructions will be in the format of:

```
git clone http://urltoproject/repository-name.git
```

optionally, you may include a new directory name (by default the name of the project is used as the directory name).

```
git clone http://urltoproject/repository-name.git new-directory-name
```

Lesson Objectives

By the end of this lesson, you will be able to clone a repository from a code hosting system, such as GitLab.

Self-Check

Do you have a copy of the project on your hard drive? Navigate into the directory and make sure there are files present. There will also be a hidden folder, `.git`.

Summary

The commands we used in this lesson allowed us to create a local copy of our GitLab project.

```
git clone http://urltoproject/repository-name.git new-directory-name
```

We can repeat this command as many times as you like, to download the repository multiple times (this can be helpful if you want to throw away your local copy and start again from scratch).

Reviewing History with git log

We'll be working a lot more with Git's log command in the lesson on finding and fixing bugs. For now, let's use it to get a real quick overview of the repository we've just downloaded. We can use Git in two or three different ways, depending on what's compiled into our system. We'll look at the full commit messages, a short log, and (if you have it installed) Git's graphical browser. We'll also take a quick look at how this compares to how GitLab displays the information.

Lesson Objectives

By the end of this lesson, you will be able to review the history for a local repository using the log command.

Self-Check

Are you able to find the two most recent commit messages for the downloaded repository? What was a quick summary of the changes made? Who were they made by?

Summary

In order to review the history of our repository, we need to use the log command. There are a number of parameters we can use to show slightly different information.

- `git log`
- `git log --oneline`

Finally, the last little tool I find helpful, is some kind of graph generator for my repository's history. This tool gives me the context of where a change came "from" (are you a branch? is it ahead of another branch in terms of commits?). GUIs for git will give you a much more elegant view of your repository than these tools, but they work just fine for me.

I can either draw the graph at the command line:

`git log --oneline --graph`

An example of this output would be as follows:

```
* edcd486 Updated deps.
* 05af892 No need for layout anymore.
*   eb85b12 Merge pull request #11 from stevector/master--implements-spelling
|\
| * 086b01e Spelling fix in 2013-02-04-highlight.md
* |   c5f46bd Merge pull request #10 from stevector/master--fixing-typo-in-readme
|\ \
| * | 58e67f4 Spelling fix in README
|/ /
* |   eebb37d Merge pull request #8 from kenjis/fix_url
```

The dots represent a commit. Anything to the right of the first is (or was) a branch. The branch names are not listed in this output.

Or, I can open up the git browser, gitk, and click around a bit. The git browser is a special add-on which might not be enabled in your version of git. For example: this functionality is *not* available in the OSX-installed version of git, but it is available in the brew-installed version. To open the browser, use the following command:

`gitk`

Configuring Git

Over time, you will find little shortcuts that help you use Git at the command line. Personally I've found those who are the most frustrated with it, are the ones with the least amount of customization. You don't *need* do to any of the things I've setup in this lesson, but you might find them a little helpful. As this setup will be hugely dependent on how you are working at the command line, I'm not going to spend a lot of time in the *video* showing you how to get this setup. There are some resources in the repository that you've already downloaded (look for the "sample" files in the directory `resources`).

There are two types of configuration settings you'll be making when working with Git: global settings which apply to all repositories that you work on, and local settings which only apply to the current repository. An example of a global setting might be your name; whereas your email might be customized based on personal projects and work projects.

Global settings are stored in in the file `~/.gitconfig`, local settings are stored in the file `.git/config` for the specific repository you are working in. You will always be able to go back and edit your settings if you want to.

Identifying Yourself

In order to get credit for your work, you'll need to tell Git who you are. We'll store this setting globally. As it's a global setting, you don't need to be in a specific repository to make the change.

Lesson Objectives

By the end of this lesson, you will be able to add your name and email to Git for proper attribution in commit messages.

Self-Check

When you run the following command, is your name displayed?

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

Summary

```
git config --global user.name 'Your Name'      git config --global
user.email 'me@example.com'
```

If you want to make these changes only to the current repository, replace the parameter `--global` with `--local`.

Changing the Commit Message Editor

By default, you'll be using Vim. I really like Vim, so that's what you'll be seeing in the videos. It's a bit hardcore though, so you might want to change your editor to something else. Note: the commit will only finish when you QUIT the editor, so you'll want to use something fairly lightweight.

Lesson Objectives

By the end of this lesson, you will be able to change the default text editor for commit messages.

Self-Check

When you run the following command, is the name of your text editor displayed?

```
git config --get core.editor
```

Summary

To change the text editor for your commit messages, use one of the following commands as is appropriate for your editor of choice:

```
git config --global core.editor mate -w        git config --global
core.editor subl -n -w
```

Adding Color

Reading huge walls of text can be difficult. We'll add some color helpers to our command line to make it easier to see what git is doing.

Lesson Objectives

By the end of this lesson, you will be able to enable the color hinting to show branch colors, and diffs by color.

Self-Check

Within the repository you downloaded for the previous lesson, when you view the log, are the commit hashes a different color than the author, date, and commit messages?

```
git log
```

Summary

```
git config --global color.ui true
```

Lesson Bonus: Customize Your Command Prompt

If you're working from the command line, you get ZERO clues about what's going on with your files, until you explicitly ask Git about them. This is tedious to keep having to ask. It's like when you were 8 and sat in the back of the car whining at the driver saying, "are we almost there yet?"

Instead of having to explicitly ask, I've modified my command line prompt to tell me which branch I currently have checked out and whether or not I've made changes to any of the files in my repository. This is a fairly common hack, but every developer will have their own little quirks on how they implement it. Searching the web for `bash prompt git status` will yield lots of results. My own prompt is fairly simple, but others have added a lot more details to their prompt. For example: Show your git status and branch (in color) at the command prompt or local file status. As with all things technical: the more you add initially, the more you'll need to debug if it doesn't work right away.

I found the fancy prompts to actually be quite fussy to set up, and ended up giving up on the really detailed ones. I recommend you too start with something really simple and then add to it if you *really* need more information. The simple change in colour, along with the name of the branch, actually suits me just fine and is less distracting without all the extra information.

This as a self-study piece.

Getting Started as a Team of One

In the previous lesson we took a look at a repository which already existed. For many people this will be a typical first experience with version control: working with something that someone else started. Eventually though, you will have a need to setup your own repository. In these next few lessons, we'll cover a few of these scenarios.

- Initializing an emptry project
- Converting an existing set of untracked files to git

Initializing an Empty Project

Use the command `git init` to start a new project locally. This creates an empty directory that you can now put stuff into.

Lesson Objectives

By the end of this lesson, you will be able to begin a new project which will be tracked by Git.

Self-Check

By the end of this lesson, you will be able to create a new git repository.

Summary

- `git init`
- `ls -al` (or on Windows `dir`)

Converting an Existing Project to Git

In the previous lesson you started an empty repository. Once you've created the repository it's time to get to work and add files to the repository.

You can start with any folder of files and create a git repository from it, using the `git init` command. This time instead of starting with an empty directory, start with an existing directory of files and "convert" your non-versioned project into a git repository.

Once the repository has been created, there are two basic operations you need to run to add files to the repository:

- `git add`
- `git commit`

When you `add` the files to the repository, you're actually added them to a temporary staging index. This temporary stage allows you to orchestrate, or direct, the specific scene you want to commit to the repository. This means you can work on bigger problems, and then segment your thinking into individual commits. When you first start working with Git, this can seem like an unnecssary step, but as you become more comfortable with the advanced functions in git, you see the advantage of being able to craft commit messages. (We'll talk about this in later lessons.)

Lesson Objectives

By the end of this lesson, you will be able to add files to a repository.

Self-Check

Make sure your files were added to the repository by ensuring there are no untracked files in the directory, and you have a log message. Use the commands `git status` and `git log` to confirm your work.

Summary

- `git init`
- (add files to the directory you've just initialized)
- `git status`
- `git add`
- `git status`
- `git commit`
- `git status`

Overview of Connecting to Remote Repositories

You've been working locally, and suddenly you realize you could share your work with the upstream project. Depending on how you started your project, you will need to connect your local work in slightly different ways.

In a centralized version control system, like subversion, there is one master copy of the repository and all work is written into that copy. When you commit, the information is immediately uploaded to that central repository and available to others. In a decentralized version control system, like Git, there is no single repository that everyone works with. It is merely a convention which declares one copy of the repository to be priviledged (and considered to be the official source for the code).

In this next set of lessons, we'll explore what it means to connect to a remote repository, and how do make a connection under a few different scenarios.

Copying a Repository

Previously you learned how to clone a project, this hooks up the remote server as a connection to your repository. You can also start with any folder which has a .git directory in it. For example: you could share your git repository with someone via a zipped email (AVOID doing this with a tool that does synchronization, such as Dropbox, as it will corrupt your git repository!!).

Making a copy of the repository in this way will make an exact duplicate of everything, including the history of where this repository started from.

Lesson Objectives

By the end of this lesson, you will be able to make copies of your git repository, and describe the limitations of using this system.

Self-Check

After making a copy of your repository, can you add a new file to it? Confirm this by reading the log messages for your repository after adding the new file.

Summary

You can easily make a copy of any respository in its current state by simply making a copy of the directory. When I was first learning git, I often took advantage of this feature to throw out all of my uncommitted work and start again from the last commit in the repository. (By the end of this series you'll have already outgrown this trick!)

The disadvantage to this system is that the copy still has YOUR remote(s) defined, and what you really want when you're sharing with someone else, is to have a third "code hosting" remote repository set up. You can check to see which remote(s) are defined by your repository with:

- `git remote`
- `git remote -v` will give you a bit more information
- `git remote show [name_of_remote]` will give additional details about the remote (remotes are typically somewhere else, you will need an internet connection when using `remote show`)

Cloning a Local Repository

You want to play with remotes, but you're entirely offline (for example: watching these videos behind a firewall and you can't get access to GitLab, GitHub, etc). Use a local directory as your "remote" repository. Remote actually just means any directory that's not THIS directory.

Lesson Objectives

By the end of this lesson, you will be able to clone a repository to the same file system.

Self-Check

There is a new copy of your repository, and when you view the remotes for the new repository, it lists a location in your file system as the remote. For example:

```
origin  /Users/emmajane/Git/workshop-git-for-teams (fetch)
origin  /Users/emmajane/Git/workshop-git-for-teams (push)
```

Summary

- `git clone [/full_path/to/repository] [new_project_folder_name]`
- `git clone /Users/emmajane/Workshops/git-for-teams local-clone`

Note: the folder `.git` is in the root of the folder `git-for-teams`.

Converting a Set of Files to a Repository

Initially you didn't think you would be doing much with a project, so you downloaded a zip folder with all the project files instead of cloning the project. (in the previous example you DID have the folder `.git` included in set of files). You now want to submit a contribution to the project, but you have absolutely no local history from the upstream project to connect with.

As we did previously, the first thing you'll need to is fork the project. This will give you a writable copy of the repository so that you can upload your changes, and then submit a pull or merge request back to the project (you'll learn about pull requests later).

Once your fork is setup, clone the repository to your local computer. Copy your editing files into the repository. Add the files to the changing area, and make a commit. If you want to be really fancy, you can do a bit of extra homework and lookup "interactive commits". This will let you, by file and by hunk, decide what should be included in a commit (the default is to work at the file level).

Note: this should ONLY be done if you have one or two minor changes. If you've been working on your changes for a long time, it's quite possible your zip copy is out of date. Copying all the files into the repository WILL introduce regressions and undo any work that has been made on the project since you last downloaded it. So be careful, eh?

Lesson Objectives

By the end of this lesson, you will be able to combine a set of previously untracked files with a repository.

Self-Check

Use the command `git log` to ensure the changed files were added to your repository.

Summary

- `git clone`
- copy the changed files into the repository
- you can use `diff`, or a difftool (I like Kaleidoscope) to show you the difference between the two sets of files
- `git add .`
- `git commit -m "Detailed message about the work you've done."`

Adding Another Remote Connection

You initially cloned a repository that uses a forking repo. Now you want to publish your results, while still maintaining a connection to the the upstream project. To accomplish this, you will need to add a second remote to your project.

I do this all the time when I'm working with open source projects. I think that I'm never going to want to make a contribution to the project, and then I find a little bug which I need to fix for myself, and decide I want to contribute back to the project. I've also added additional remotes when I want to have a backup, or if the code hosting has changed for some reason (e.g. I don't have access to a client's code hosting system yet, so I upload to a private repo on my code hosting).

In the next set of lessons we'll talk about branching, and keeping these remotes up-to-date.

Lesson Objectives

By the end of this lesson, you will be able to add additional remotes to your repository.

Self-Check

When you run the following command, you should see two lines (one remote):

- `git remote -v`

results in something like this:

```
origin  git@github.com:emmajane/vagrant-solo-dev.git (fetch)
origin  git@github.com:emmajane/vagrant-solo-dev.git (push)
```

After adding a second remote, the output of `git remote -v` should list another two lines (fetch and pull).

```
dme git@github.com:DrupalizeMe/vagrant-intro-series.git (fetch)
dme git@github.com:DrupalizeMe/vagrant-intro-series.git (push)
origin  git@github.com:emmajane/vagrant-solo-dev.git (fetch)
origin  git@github.com:emmajane/vagrant-solo-dev.git (push)
```

Summary

- `git remote add [name] [url]`
- `git remote show`
- `git remote -v`

Working with Branches

In version control, branches are a way of separating different ideas. They are used in a lot of different ways. You may use branches to denote different versions of software (for example, Drupal a branch for version 5, 6, 7, 8, 9 and each of these branches contains significantly different code). You might use very short term branches to work on a new bug fix. You might use a branch to test out a new idea.

If you're not sure if you should be working on a different branch, it may help to ask yourself a few questions:

1. Is it possible I will want to completely abandon this idea if things don't work out?
2. Am I creating something which is a significant deviation from the current published version of the software?
3. Does my work need to undergo a review before it's published, or accepted into the published version of the software?

If you answered "yes" to any of these questions, you should consider creating a new branch for your work.

Within a branch, you may have one or more commits. You were already introduced to the concept of a commit when you added new files to your repository in the previous lessons.

A commit is a unit of intention which you may want to reverse, or build from at a later date. When you first start working with distributed version control, chances are good you'll work in commits which are actually too granular for advanced tools (you'll learn more about this in the lessons on rebasing). This is because we're coming from a different mindset when it comes to "saving our work" and being able to "undo mistakes". When we use this mindset, we think of clicking the save button, or using undo to remove the last few things we typed. When the commits are this small, the commit messages tend to be nearly useless ("stopped for lunch"; "tried something"; "didn't work"; "ooops"; "testing"). If we wanted to rollback history, how the heck would we use those commit messages to find the spot where the code was working the way we intended!

It took me quite a while to get OUT of the habit of thinking of Git as a place where I "saved my work" and instead as a place where I "recorded my results". As you develop your sense of what makes a useful commit, simply separate your work into different branches. As you confirm the work is done, you can merge this completed branch back into the main development line.

In these lessons you'll learn about the strategies software development teams typically use to separate branches; you'll learn how to work with branches; and we'll walk through a typical example of how to work with a branch on a remote repository. Unlike many git tutorials, I'm going to focus on how we use branches when *collaborating with others*, and not just working on our own project (much like I am with this repository of lesson notes).

Listing All Branches

Branches allow us to work on independent thoughts at a time. Your repository will have its own branches. Remote repositories have their own branches. Just because you can see a branch, doesn't mean you have the latest content.

Lesson Objectives

By the end of this lesson, you will be able to list local branches, and remote branches for your repository.

Self-Check

Using a cloned copy of the the lesson repository, you should be able to list the local branches, and the remote branches.

Summary

- `git branch`

- `git branch --list`
- `git branch -a`
- `git branch -r`
- `git fetch`
- `git remote show <remote_name>`

Using a Different Branch

When you checkout a branch, you are updating the visible files on your system ("the working tree") to match the version stored in repository.

When you switch between branches, it can sometimes be helpful to have a Finder window open so you can see what files are changing (appearing and disappearing) when the different branches are checked out.

Lesson Objectives

By the end of this lesson, you will be able to check out a local branch.

Self-Check

When you changed to a different branch, the git log showed a different commit for the most recent work than the branch you were on previously. (You can check this with `git log --oneline`.) Technically two branches don't NEED to have different commits though, so self-check isn't perfect.

Summary

- `git checkout <branch_name>`
- `git checkout --track <remote>/<branch_name>`

Establishing Your Branching Strategy

As you build up your work, you'll add more parallel branches to your work flow. Any work which happens in a branch is completely isolated from other branches in your repository. Can you have completely different files for completely different versions of your software. The branches should all have a relationship to one another though, they shouldn't contain completely different projects!

You can invent your own branching strategies. For example, in the repository for these lessons, I have one main branch that I use for workshops. Nearly all of my work is done directly in this branch and it is immediately visible when people visit the project page.

I also have a few branches for ideas that are still in progress, and not ready for immediate consumption (they're still available, you just need to know to look for them). For example: while I was working on the video lessons, I created a new branch. When the lessons are recorded, I'll merge this branch back into the master branch so that the content is immediately visible to everyone. I also have a branch with random files which I have no intention of merging back into the master branch. This branch, sandbox, is meant to help you learn about branching. It's sort of a fun Easter egg for people who aren't taking the lessons, and don't bother looking into the branch.

What I'm doing with this repository is quite specialized and unique to my project. You can invent your own branching strategies for your own projects, but when it comes to software development, there are a number of "best practices" out there which you may want to consider following. By using a convention, you make it a lot easier for other people to collaborate on your project, AND do things correctly.

Let's take a look at these branching strategies now.

There are effectively two ways that teams can work on software: either they collate the work, saving it up for a major release; or they are continuously deploying very tiny changes to production. (I sometimes joke that the original continuous deployment was live editing on the server. Some people think I'm being serious when I say this though … so be careful if you try to make the same joke.)

Scheduled Release

- Most popular example: GitFlow.
- Optimized for the collation of many smaller changes into a single release.
- Typically used for a download-able product; or web site with a scheduled release cycle (e.g. "Wednesdays").
- Incorporates human-reviews, and possibly automated tests.

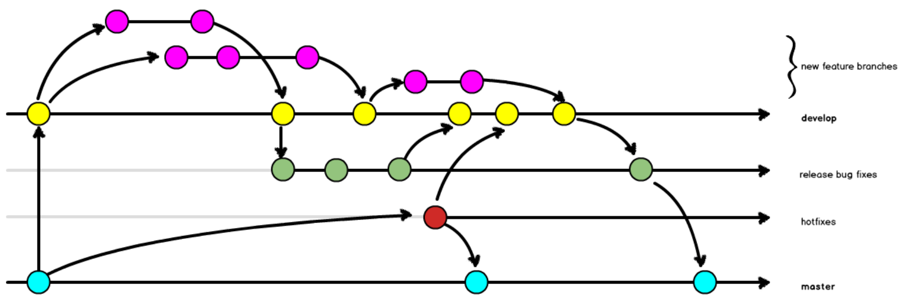

Figure 1: scheduled release: gitflow

If you have the concept of stable releases, hotfixes, point releases, security releases, multiple supported versions, etc, then you need this granularity for your branches.

There is always a period of time where you do not trust your code/developers and want to have a separate QA period. Thinking like a download-able product: version 4 vs. version 5 of The Software (a piece of software).

Continuous Deployment

- Code is deployed faster than scheduled releases; assumes all check-ins are deployable.
- Requires (trusted) test coverage.
- Typically uses a mechanical gatekeeper (CI) to check in code to the master branch.
- Often has flippers/flags for fine grained access to in-progress features.
- Fewer branches to maintain / keep updated.

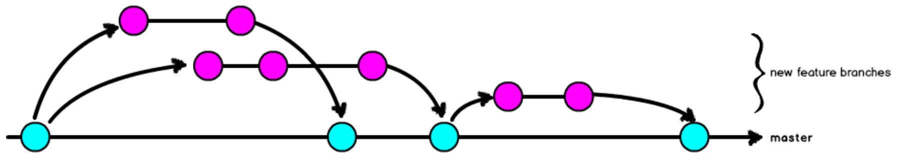

Figure 2: continuous delivery: feature branches

If you don't need the granularity of multiple supported versions, you can probably get away with something closer to this branching strategy. Can you get away with just tags? Do you intend to go back and work on a previous version? As soon as you have the concept of a separate security hotfix, you need to introduce a separate branch. In CD: everything is urgent, so there's not a separation of a really urgent security fix. CI, CD vs CD: http://puppetlabs.com/blog/continuous-delivery-vs-continuous-deployment-whats-diff

A great way to familiarize yourself with different branching strategies, is to check out what projects are doing. Here are a few to check out:

- `git clone http://git.drupal.org/project/drupal.git`
- `git clone https://github.com/git/git`
- `git clone https://github.com/twbs/bootstrap`

Lesson Objectives

By the end of this lesson, you will be able to identify two common branching strategies, and explain under what circumstances they are appropriate development strategies for teams.

Self-Check

Using the diagram you created for the warm-up exercise, can you plot onto this diagram where the potential branches might be? Does your current setup look more like a scheduled release (GitFlow), or continuous deployment branching pattern?

Summary

These days continuous deployment is the new hot. Everyone wants to be able to do this. And for very tiny projects, I definitely just work in master and putter along with each commit acting as a resolution to a problem; however; the bigger the team gets, the more it will benefit from having some structure in how the people collaborate on the work.

New teams are unlikely to have the infrastructure needed for continuous deployment. They are less likely to be writing reliable tests, and they are more likely working on longer sprint cycles (with deployments happening once every couple of weeks, not several times a day). For this reason, this video series is going to focus on how to work with your team in setting up a scheduled release branching strategy.

Creating a Topic Branch

It's very easy to create a new branch in Git. Almost too easy…as Git doesn't provide any guidance on what you're supposed to do in that branch. Assuming you're working on a software project, the best way to decide what goes into a branch, it start with the Issue Tracker.

by creating a written description of what you're about to do, you will have a clear sense of when to start AND finish with your branch. Yes, this will often feel like overhead, but it is a really good habit to get into, especially when you're working in larger teams.

In GitFlow, all new ideas use "Feature" branches. (There's nothing special about them technically, it's just a naming convention to make it easier to think about what kind of work happens in the branch.) Feature branches can incorporate one, or many tickets if the idea is really big. Be careful though, if you have a lot of people working on really different ideas, and their branches diverge significantly, you risk creating mini projects instead each repository, making it difficult to incorporate the work into the common branch at release time.

If it's a new project, you may only have feature branches, and ane integration branch. By the GitFlow convention, your integration branch would be named `develop`, and the master branch wouldn't be updated until you had an official release. Do whatever works for you, but make sure you document whatever you do!

In your private repository on GitLab, create a new issue ticket which outlines the changes you're about to make. Next, create a corresponding branch in your repository. Do your work. Finally, save the changes to your local branch by adding the changed file, and then committing it.

Lesson Objectives

By the end of this lesson, you will be able to create a new branch using best practices, to solve a specific issue identified in your code hosting system.

Self-Check

You have a new branch created using the naming convention `issue_number-terse_description`.

Summary

1. Create a new issue; note the number on the issue

The new ticket I created was as follows:

```
Problem: The repository is too serious.

Rationale for change: having more jokes in the repository will allow people to
take a break from learning git and have a bit of a laugh.

QA: After this ticket has been resolved, I will be able to read more bad jokes.
```

2. Create a new branch using `git checkout -b <issue-description> sandbox`
3. Add a new joke to the file `badjokes.md`. You now have a "dirty" repository with one modified file.
4. Add the joke file to the staging index with `git add <file>`.
5. Commit your joke file to the repository with `git commit`.

I usually do a commit at the command line, and then commit –amend to write a longer commit message which relates to the issue.

Uploading Your Changes with git push

To upload your changes to the remote repository, we'll use the command `push`. A few things need to be in place for this to happen though.

1. You need to have write permission to the repository. Assuming you setup your SSH keys correctly, this shouldn't be an issue.
2. The branch you are uploading needs to have a connection to a specific remote repository. i.e. you need a "destination"

I generally take the lazy approach to pushing up my branches, and always start with the shortest sequence, adding parameters as needed.

Lesson Objectives

By the end of this lesson, you will be able to upload your branches to remote repositories by using the Git command push.

Self-Check

Log into GitLab to ensure the branch has been uploaded to the remote repository.

Summary

`git push` `git push <remote_name> <branch_name>`

Accepting and Merging New Work

Once the code review process has been completed, it's time to accept the new work into the main branch for your project. There are a few different ways you can combine branches in git. The first maintains the branch identity, showing that at one time these commits lived in their own branch. The second does not maintain branch identity. Instead, it fast-forwards through the commits you're adding to the main branch. In the history, the fast-forward merge (which is the default) will make your historical graphs look as though the commits have always been in place. Those who use a rebasing workflow will often choose to merge branches with `--no-ff` to maintain the illusion of the branch, while always updating their master branch with a rebase to keep a perfect set of "swim lanes". If you prefer a history with fewer lines to follow, use the default.

Lesson Objectives

By the end of this lesson, you will be able to update a update a local branch using git pull, and merge a local development branch using the fast-forward and true merge strategies.

Self-Check

Are you able to incorporate the sandbox branch, or the joke branch into your work?

Summary

- `git checkout master`
- `git fetch`
- `git merge <remote>/master`
- `git pull` // a more aggresstive shortcut to fetch + merge
- (assuming master is now up-to-date)
- `git merge <ticket_branch>` // attempts fast forward merge
- `git merge --no-ff <ticket_branch>` // forces a new commit, maintains the illusion of a branch
- `git push` // upload the revised copy of the master branch

Example of fast forwarding vs. a true merge:

- `git checkout -b integration_ff`
- `git merge 1-bad_jokes`
- `git log --oneline --graph`
- `git checkout -b integration_no-ff master`
- `git merge --no-ff 1-bad_jokes`
- `git log --oneline --graph`

Dealing with Merge Conflicts

If two developers have changed the same part of a file and you try to merge those changes, Git will be unable to know which one should be kept and will stop the merge process. You will need to check the files by hand and determine which copy to keep. You may want to use a mergetool for particularly complicated merges as it will give you a nicer reference to make decisions from. I personally use Kaleidoscope, but there are other free tools for each of the different operating systems.

Once you've solved a merge conflict once, you'll never want to have to solve it again. You can use `rerere` to save your resolutions.

Lesson Objectives

By the end of this lesson, you will be able to resolve a merge conflict.

By the end of this lesson, you will be able to enable to “RE-use a REcorded Resolution” while rebasing.

Self-Check

Edit two files at the same place in two different branches and then try to merge them. Decide ahead of time which of the two branches you want to keep the changes from.

Add the same error (which generates a merge conflict) to two different branches, and bring them both up-to-date with rebasing. You should only have to resolve the first merge conflict.

Summary

- `git config --global rerere.enabled true`
- `git checkout -b one`
- (edit the readme title)
- `git checkout master`
- `git checkout -b more`
- (edit the readme title)
- `git merge one`
- (edit the file with the conflict)
- `git add .`
- `git commit`

Working with Tags

Tags are used to pin-point specific commits. They don’t follow a series of commits, like a branch does. They are a single point in time along the history of your work. You can think of them like a bookmark on your work. I don’t use tags nearly as much as I should. As a result, I rely on my commit messages to find specific points in the repository. You may find working with tags is a good habit to get into as they’ll allow you to easily reference points in your time line.

Listing, Adding, and Deleting Tags

Lesson Objectives

By the end of this lesson, you will be able to view a list of tags, and add a new tag.

Self-Check

When you start this lesson, there are no tags on the repository. You'll need to add some tags before you can view them.

Summary

- `git tag`
- `git log --oneline`
- `git tag -a <tag_name> <commit_id>`
- `git tag -a -m "Reason for tagging" <tag_name> <commit_id>`

Checking Out Tags

Once a tag is made, you can investigate just the commit where the tag was added.

You can also checkout this tag. You'll go into a DETACHED HEAD STATE! But you'll be fine.

Lesson Objectives

By the end of this lesson, you will be able to investigate a single commit which has been tagged.

By the end of this lesson, you will be able to locate and checkout a specific commit hash by using a tag as the reference point.

Self-Check

You should be able to view the tag message, commit message, and the diff for that commit.

When you checkout a tag, you'll get a warning message about being in a DETACHED HEAD state.

Summary

- `git show <tag_name>`
- `git checkout <tag_name>`

Recovering from a Detached HEAD State

Checkout the branch you were on previously to return to your regular work.

Here's an example of the message from a checkout I did:

```
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 performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 255a271... Lessons were renumbered to match O'Reilly conventions.
```

Lesson Objectives

By the end of this lesson, you will be able to describe what a detached HEAD state is, and be able to recover from it.

Self-Check

You can get rid of the error messages about being in DETACHED HEAD.

Summary

- `git checkout <tag_name>`
- `git checkout <branch_name>`
- `git checkout <tag_name>`
- (make changes)
- `git checkout -- <filename>`
- `git status`
- (make changes)
- `git checkout -b <new_branch_to_save_work>`

Sharing Tags

Tags are only available on your local repository unless you explicitly share them with others.

Lesson Objectives

By the end of this lesson, you will be able to add and remove tags from your remote repository.

Self-Check

Are you able to find your tags on the remote repository?

Summary

- `git push --tags <remote_name>` (add the –tags parameter when pushing branches)
- `git push origin :<tag_name>`

Finding and Fixing Bugs

There are two ways to look for bugs: determining where the problem code is; and determining when (and by whom) the problem code was introduced. Generally I debug from within the current code, but in more complicated code bases it can be useful to compare "working" and "non-working" states (we'll use git bisect for this).

Finding Relative History with Git Log

Tips and tricks for getting the most useful git log messages – match this with diagrams of what the command is actually showing.

Lesson Objectives

By the end of this lesson, you will be able to choose very specific ranges of history to look at.

Self-Check

Draw the graphs for what each of these combinations represent. Check your work using gitk to verify your diagrams.

Summary

- `git log HEAD^` // not including the latest
- `git log HEAD~3` // not including the last three
- `git log <branch>..<branch>` // difference between two
- `git log <commit>..<commit>`
- `git log <branch>..` // what's here, but not there
- `git log ..<remote/branch>` // what's there, but not here
- run all of these again but with `git diff`, not `git log`

Finding the Last Working State with Bisect

Often it can be difficult to figure out exactly how a bug was introduced in your code. You know something went wrong, but you just can't figure out where that extra 2px margin came from. Introducing git bisect!

First you'll need to define the problem you're trying to isolate, and then you can use git bisect to find the commit where the problem was introduced.

If you need an emergency "out" of the bisect you can simply use checkout to return to a different branch. This is much like recovering from a detached head state.

Lesson Objectives

By the end of this lesson, you will be able to use git bisect to find a specific state in your code.

Self-Check

Were you able to find the bug?

Summary

- `git log <since_last_merge_to>..<what's_been_added_here> --oneline`
- `git bisect start`

- `git bisect good <commit_id>`
- `git bisect bad <commit_id>`
- `git bisect reset`
- `git checkout <branch>`

Gotchas

- You need to be in the top-level directory to run bisect.
- It is assumed that the current work is “bad”. So, you can’t go back and find when something is fixed, you need to go back and find where something broke. (Git gets very confused if you try to find where a **fix** was introduced.)

Finding the History of a File with Blame

Quick command to get a history of who has changed a specific file. Can be helpful when you’re working in teams to find the person who might know more about the specific changes that were made to that part of the code base.

Lesson Objectives

By the end of this lesson, you will be able to look at the history for a specific file.

Self-Check

Summary

- `git blame <filename>`
- `git log <commit> -p`

Using Stash to Work on an Emergency Bug Fix

You need to do something “real quick” for a client, but you’re in the middle of a complicated set of tasks in your current branch. Put all of this work on hold with the “stash” command instead of writing a useless commit that you have to unpack later.

I once heard someone joking about how he could “git stash his morality” when watching horror movies. In other words: he could put his values aside while he watched the movie. … maybe you needed to be there.

Lesson Objectives

By the end of this lesson, you will be able to put your work on hold temporarily with git stash.

Self-Check

Summary

- `git stash`
- `git stash list`
- `git stash apply`
- `git stash drop`

Rollbacks, Resets, and Undoing Your Work

There are a lot of ways to undo work in Git, and each method is exactly right sometimes. Unfortunately the commands can be a little confusing to use as they're so closely named. You might want to make a cheat sheet for yourself. In fact, I highly recommend creating little sketches for yourself as you learn these commands. Creating your own drawings will help reinforce the learning, and it will give you a mental reference point, which is often easier to remember than a manual page.

Instead of explaining each command git offers, I'm going to give you a few different scenarios when you might want to undo your work.

- Doing experimental work in new branches.
- Deleting unsaved changes in the working directory.
- Rewinding time by removing changes (and the commit message).
- Time travelling, by going back to an old commit.

In addition to these methods, rebasing allows you to unhook previous commits from a chain of events, and create a new chain in the history (without actually deleting anything). We'll be covering rebasing in the next lesson.

If you've committed something into the repository, it will always be there. It's virtually impossible to actually delete work in Git. It is possible, however, to lose work and not be able to find it again. If you think you've *lost* history, you'll want to use reflog to find your lost time.

Using Branches for Experimental Work

Previously we've created, and deleted branches, using the ticket as a starting point. But what if you were working on a ticket, and you weren't sure which of two approaches you should take. In this case it would be absolutely acceptable to create a branch off of your ticket branch, noodle around some bit, and then bring those changes back into your ticket branch.

Lesson Objectives

By the end of this lesson, you will be able to create, use, and delete a new branch.

Self-Check

You can checkout your new branch, and remove it from the list of branches when you are finished working on your ideas.

Summary

- `git checkout -b experimental_idea`
- (do work)
- `git add .`
- `git commit`
- `git branch -d experimental_idea`
- `git push <remote> :<branch_name>`

If you want to rollback to a previous commit temporarily:

- `git checkout -b backup_branch`
- `git checkout <branch_to_rollback>`
- `git log --oneline`
- `git checkout <commit_id>`

Amending a Commit

In a previous lesson we amended a commit so that our branch would be correctly attached to a ticket we were working on. So long as you haven't pushed your branch yet, you can also use commit to add additinal work to the commit.

If you have already pushed the work it's considered bad form to go back and "fix" shared history. Make sure you're only changing your own history, or ticket branches which only you are working on.

Lesson Objectives

By the end of this lesson, you will be able to amend a commit to add new work.

Self-Check

You can make additional changes to your repository without a new commit point being added to your Git timeline.

Summary

- `git add .`
- `git commit --amend`

Removing Changes to the Working Directory

You're working along and you just deleted the wrong file. You actually wanted to keep the the file. Or perhaps you edited a file that shouldn't have been edited. Before the changes are locked into place (or "committed") you can "checkout" the files. This will restore the contents of the file to what was stored in the last known commit for the branch you're on.

Lesson Objectives

By the end of this lesson, you will be able to restore a file to the state it was in in the most recent commit.

Self-Check

You can restore a deleted file in the working directory.

Summary

- `rm README.md`
- `git checkout -- README.md`

If you've already staged the file, you'll need to unstage it before you can restore it.

- `rm README.md`
- `git add .`

- (git checkout – won't work)
- `git reset HEAD <file>`
- `git checkout -- <file>`

If you want to restore all of the files to the previous commit, you don't need to make the changes one at a time, you can do it in bulk with:

- `git reset --hard HEAD`

Removing Commits with Reset

Use this to throw away commits you didn't want to keep. When you reset your work, git throws away the commit message, but returns your working directory to the state it was in the moment before the commit was stored. So if you didn't really mean to delete a file in a previous commit, you can reset your working directory.

(It's actually a tricky one to explain, but relatively straight forward to see it in action.)

Note: this is going to alter history as it removes references to commits. This means it's best to do this only in your own history, but not in a shared history.

Lesson Objectives

By the end of this lesson, you will be able to restore your working directory to a previously committed state using git reset.

Self-Check

You can restore a file that was deleted, and remove the reference to it being deleted.

Summary

- `rm README.md`
- `git add .`
- `git commit -m "Removing a file that shouldn't be removed"`
- `git log --oneline`
- (copy the commit ID for the commit BEFORE the one you want to reset)
- `git reset <commit_id>`
- `git status`

- `git checkout -- <file>`

If you want to undo your undo, you can do that too.

- `git reflog`
- (look for the line above "reset: moving to"; it will be something like "checkout: moving from master to).
- `git show <commit_id>`
- (confirm it's the right ID to restore)
- `git checkout <commit_id>`
- `git checkout -b restored_branch`
- `git checkout master`
- `git rebase restored_branch`
- `git branch -d restored_branch`

Promoting a Previous Commit with Revert

If there was a commit in the past which was incorrect, you can reverse that single commit. Unlike reset, revert does nothing to the commits in between. It simply puts back the changes you made in the commit you're reverting. Unlike reset, revert applies the changes so you are left with a clean working directory. Resets need to be paired with a commit as you are "resetting the working directory".

Lesson Objectives

By the end of this lesson, you will be able to undo previously made changes by adding them back in a new commit.

Self-Check

You can reapply the changes that were made previously.

Summary

- `git log --oneline`
- `git show <commit_id>`
- (confirm this is the change you want to reverse)
- `git revert <commit_id>`

Repeat for each commit that you want to undo. (There's no way to re-apply changes in bulk while maintaining a record.)

Rewiring History with Rebase

Rebasing is often described as the ultimate Git experience. I have no problems using it, but I have to admit that it makes me a bit uncomfortable. Rebasing isn't about changing the results of your work (although you can if you want to), rebasing is all about changing the interpretation of how history happened, without changing the actual outcome you see in your working directory.

History revisionist complaints aside, there are a number of reasons why you would actually want to use rebase.

1. Bring a feature branch up-to-date based on changes which have happened in its ancestor branch. Using rebase in this context allows you to keep a nice clean history, and can help you avoid merge conflicts when you've finished your work and you're ready to merge it into the main line for your project.
2. Clean up granular commits and improve the commit message. This is especially useful if you've been using reset or revert and you have some odd messages in your commit history. This is also useful if you have a number of commits that, due to a peer review or sober second thought, you've decided were not the correct approach. Remember when we did the git bisect? cleaning up your history so there are only good, intentional commits will make it easier to use bisect.

Bringing Your Work Up-to-Date with Rebase

This is likely one of the first times you'll be "forced" to use rebase when working with Git. In this use case, you're using rebase to bring your local branch up-to-date by replaying previous commits which happened upstream as if they were already in place before you started your work.

Lesson Objectives

By the end of this lesson, you will be able to update a branch using rebase to add commits which have happened in the ancestor branch.

Self-Check

When you compare the two branches, ensure there are no changes in the mainline branch which aren't in your own. You can use `git log ..master` to show you commits which have been added to master. If there are none, your branch is up-to-date.

Summary

Make sure rerere is enabled!

- `git config --global rerere.enabled true`

Look to see if there are outstanding commits in the other branch, then run the rebase (it will feel similar to merging).

- `git log ..master`
- `git rebase master`
- if there are merge conflicts: resolve the conflict and follow the onscreen instructions to proceed

Using Rebase to Combine Several Commits

The next most common use for rebase is to take several little commits you've made and squash them into a single commit. This allows you to hide some of your messy work, and show only a lovely final solution to your team. This can also be used to simplify a simplifying a merge / pull request before incorporating it into the main branch for your project (the "trunk").

Lesson Objectives

By the end of this lesson, you will be able to use pick, squash, edit, and reword when rebasing your work.

Self-Check

Make a temporary branch and shuffle the last five commits around. Then verify your work by looking the history with the commands log, diff, and show.

Summary

- `git log --oneline`
- (choose a commit for the message you do want to keep)
- `git rebase -i`
- pick: leave as-is
- edit: change the files in the commit
- reword: change only the commit message
- squash: keep the commit, but merge it into the previous commit (2 for 1 deal)
- fixup: like squash, but throw-away the commit message

Using Rebase to Truncate a Branch Before Merging

This example is the same as the previous one, but this time we'll be adding the commands to merge a truncated feature branch into the master branch.

Lesson Objectives

By the end of this lesson, you will be able to incorporate interactive rebasing with merging to add a truncated set of commits to a master branch.

Self-Check

Then verify your work by looking the history with the commands log, diff, and show.

Summary

- Make a temporary branch and add three commits.
- Create an integration branch off of master.
- Merge the temporary branch.
- Rebase the commits added so there is only one new commit on the branch using `git rebase -i HEAD~3`
- Checkout master. Merge the integration branch onto master.

Alt commands:

- `git log --oneline`
- (choose a commit for the message you do want to keep)
- `git rebase -i <commit_id>`
- (set the message you want to collapse to "squash")
- (update the commit message to remove the references to the revert/reset commit)

Combining Your Changes Into Another Branch

When you merge two branches, you preserve the history that there was once a different branch. You can maintain a single line of work in your log by merging two branches, or by using rebase. If you have a specific graph you want to maintain, you need to choose the right method for how you'll combine two sets of work.

- rebase: graph looks like it's always had the commits in place.
- merge: graph looks like the merge were part of a separate branch.

Lesson Objectives

By the end of this lesson, you will be able to identify when you should use rebase, and when you should use merge.

Self-Check

You can identify from a graph if merge was used, if merge –no-ff was used, or if rebase was used.

Summary

Bringing a branch up-to-date: use rebase, or pull.

- `git rebase <branch>` // will always look like a fast forward even if there is a true merge
- `git pull` (fetch + merge with fast forwards)

Incorporating new work: use merge.

- `git merge --no-ff <branch>`

Don't really care how your graphed history looks:

- `git merge <branch>` // might look like rebase if it can fast forward, but won't if there's a true merge

Changing Previous Commits with Interactive Rebase

Let's say you tucked two files into a commit a little while ago which should have actually been two separate commits. Use interactive rebasing to go back to that point and fix what went where (note: you can also do interactive commits if you want to chunk changes in a single file into two commits).

Lesson Objectives

By the end of this lesson, you will be able to squash a reset or revert commit into the previous commit.

Self-Check

You can "remove" the reference to a commit without changing the contents of the files in the latest commit on a branch.

Summary

- `git log --oneline`
- (choose a commit for the message you do want to keep)
- `git rebase -i`
- pick: leave as-is
- edit: change the files in the commit
- reword: change only the commit message
- squash: keep the commit, but merge it into the previous commit (2 for 1 deal)
- fixup: like squash, but throw-away the commit message

Collaborating on GitHub

Overview of GitHub Introduction to who/what GitHub is. Special notes: this is a private company, it is not based on open source software. It is a platform which improves your experience with Git, but has several of its own GitHub "isms" which can make it difficult to know when you're working with Git, and when you're working with GitHub terms.

Creating an Account Signup for a GitHub account.

Forking a Repository Start by making a copy of the repository that I created. This is now your own playground to do whatever you want in. (Review of the GitLab procedure, except for GitHub.)

Making Changes to Your Fork If you want to make changes to your fork, you need to clone the work locally, and the push the revisions back up to the server.

Making Quicker Changes with the Web UI You can avoid making a local copy by using the Web UI to make simple changes to your repository. Generally it's harder to maintain good commit message quality through this method. (Linus refuses to accept commits submitted this way because of the poor formatting on messages.)

Tracking Your Changes with Issues A review of how to create a ticket with an issue.

Importing a New Repository Import the repository you created on GitLab (or download and upload the one that I created which you initially cloned from).

Accepting a Pull Request Overview of how to deal with an incoming pull request. (might need to break this into smaller segments)

Extending GitHub with Hub There is an additional set of command line tools available through the project Hub. I've used it to convert issues into Pull Requests (although I think that feature is now deprecated). Others have used it to create Pull Request builders (create a new EC2 instance for specific pull requests). If you use GitHub a lot, and are comfortable at the command line, you may find this project useful.

Collaborating on BitBucket

Overview of BitBucket Introduction to who/what BitBucket is. Special notes: this is a private company, it is not based on open source software. It is a platform which improves your experience with Git, but has several of its own "isms" which can make it difficult to know when you're working with Git, and when you're working with their terms.

Creating an Account Signup for a BitBucket account.

Importing a Repository Import the repository you created on GitLab (or download and upload the one that I created which you initially cloned from).

Making Changes to Your Fork If you want to make changes to your fork, you need to clone the work locally, and the push the revisions back up to the server.

Tracking Your Changes with Issues A review of how to create a ticket with an issue. Check to see if you can still convert an issue into a pull request.

Accepting a Merge Request Overview of how to deal with an incoming pull request. (might need to break this into smaller segments)