# Video Git Bootcamp

Video Git Bootcamp 1

Section 1: Introduction 2

(+) Installing Git 2

Optional: Building Git from Sources 2

Bare repository 2

Initial setup - configuration 3

Distributed nature of git - what is a remote 3

First commit 3

Sharing commit – git push 4

What is a commit, referring the commit 4

Getting the updates from the repository – git pull 4

Ignoring files 4

Short-hand commit syntax 4

I screwed up 1: fixing last commit 4

Section 2: Understanding git 5

Git zones: working directory, index, pushed, stash 5

Basic collaboration 5

Git as graph 5

First conflict 5

Ignoring files 5

Why clone is called clone 5

Branching 6

Basic work with branches 6

Merging the code between the branches 6

Git log – the graph version 7

Gitk – a GUI tool to explore git log 7

Deleting a branch 7

Publishing branches 7

Kinds of branches 8

Git fetch command 8

Understanding git pull. 8

Fast forwarding 9

Deleting a remote branch 9

Optional: the default branch 9

Optional: How git stores branches 9

Sharing work between branches 9

Navigating Git Graph 10

Referrencing the commits 10

Checkout and reset – overview 10

Checkout 10

What is HEAD. Reflog. 10

Reset 10

Rewriting public history 10

Reverting unpublished commit 10

Reverting published commit 10

Squashing unpublished commits 10

Undoing merge 10

Undoing rebase 10

Tags 10

Creating a tag 10

List all tags, getting code at tag 11

Deleting a tag 11

Git workflows 11

Github 11

GUI clients 11

Integration and setup 11

Other topics 11

Line endings in git 11

Why shared repositories should be bare 11

Merge two git repositories (projects) into one 11

Notes 11

I screwed up!!! Ideas. 12

Introduction: what is version control

## Section 1: Introduction

Welcome to Git BootCamp course. Unlike many other courses our goal is *not* to make you memorize a bunch of git commands or teach you to perform most common tasks. Instead, what you’ll get is a solid understanding of how *git really works*. We’ll teach you to *think like git*. With that knowledge performing even complex tasks will turn out to be a quite easy logical process.

Indeed, git is hard to memorize (my head is spinning when I’m thinking about all the options and commads that you can perform) but at the same time, it is quite easy to *understand*.

### About version control

More than ten years when I started to do my university assignment on programming I often was afraid to break the code that was already working, because I wasn’t always sure that there is a way back.

To help me feel a little bit more confident, I started to archive a project folder. Storing every archive made me much more confident. No matter how hard I screw up I always could delete my entire project and restore it from one of the older archives.

The version control system is a tool that essentially does the same thing – saving the snapshots of a project at a different times to restore it if something went wrong. Modern version control systems (VCSs) do that in a much smarter way than just archiving a whole folder, and their feature sets are much wider than restoring an old state.

Git allows to organize an effective work of a big team of people (and when I say *big* I mean hundreds) and let them easily exchange their code, switch between different versions of software, propagate fixes, and find bugs.

Besides Git is distributed, meaning that not a single developer is dependent on the server. In fact, every developer in a team will have a full history of project evolution stored on their machines and in a case of server crash it will take them no more than 10 minutes to restore the repository on another server.

### Installing Git

The easiest way to get Git is to download a pre-packaged version from <http://git-scm.com>

Check out the video for the installation details for MacOS and Windows.

**NOTE:** at the time of writing Windows users have only the access to 1.9.5 version of git, while the most recent version today is 2.5.0 that has many improvements over 1.9.x versions.

### Optional: Building Git from Sources

TODO: Make a video showing the process

<https://git.wiki.kernel.org/index.php/WindowsInstall>

<http://msysgit.github.io/>

### Initializing the repository

To start working with git you have to create a repository. There are two kinds of repositories in git:

* *Bare* repositories are used for sharing the code
* *Working copies* are used, as the name suggests, to work

Let’s create a new bare repository and then clone it to make a working copy to save our project.

Show git init --bare command and git clone. Explain that git stores all its settings in .git folder, show the folder structure (briefly)

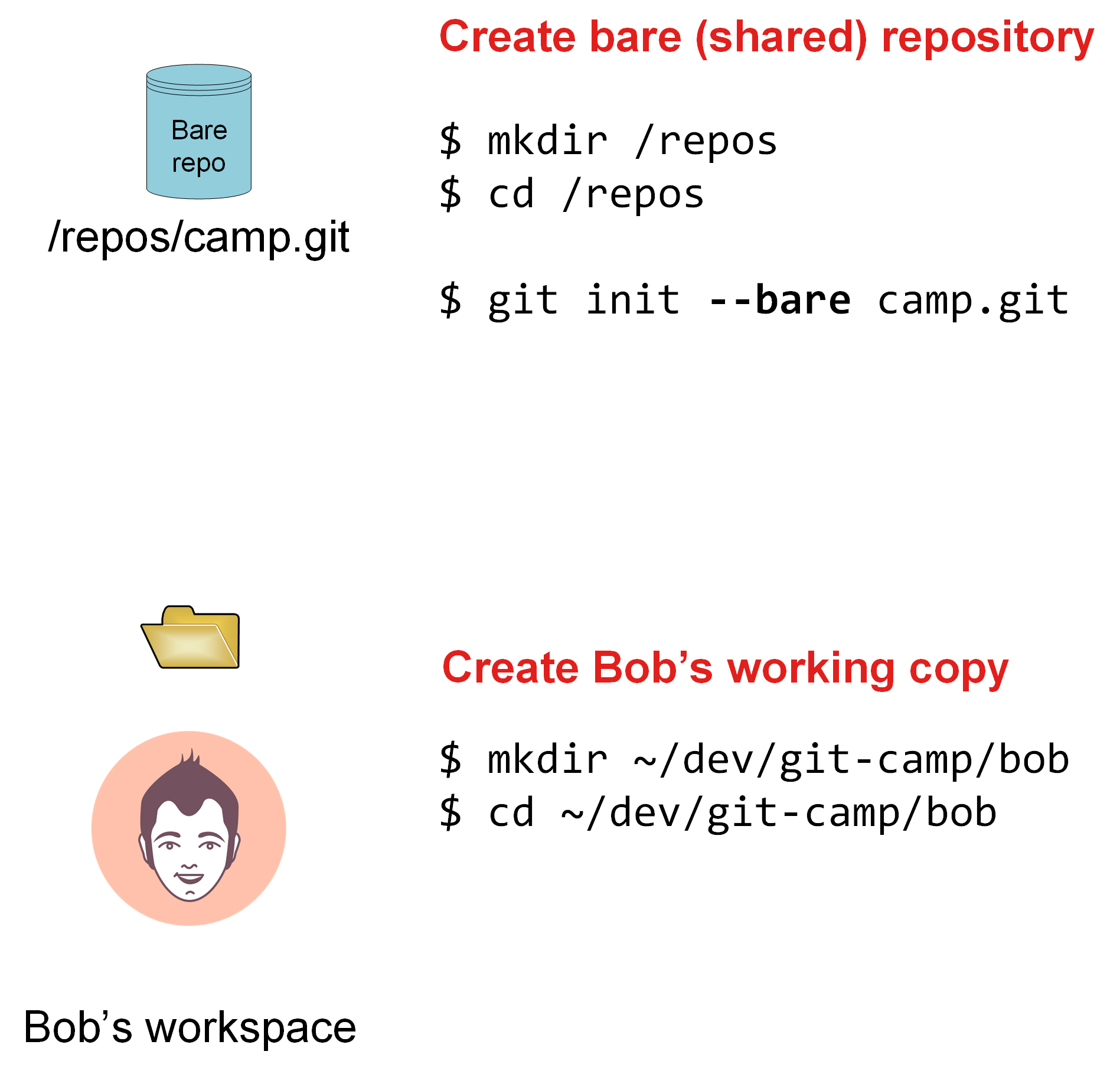
$ mkdir /Users/juriy/repos  
$ cd ~/repos  
**$ git init --bare camp.git**

Now, there’s an empty bare repository initialized under ~/repos/camp.git. camp.git is a folder, even though it looks like a file. It is used to share the code between developers. To start working with this repo let’s create a working copy by *cloning* it.

$ mkdir ~/bob  
$ cd bob  
$ git clone /Users/juriy/camp.git/ .

In the example above, ~/bob is a project folder of one of our developers – Bob. That’s where he will be adding code in the next chapters, but right now this folder has nothing but a subdirectory called .git. This is a place where git stores everythig it needs for this working copy: settings, encoded files and change history.

Image below summarizes the workflow that we just did:



**NOTE:** Good article on bare/non-bare repositories: <http://bitflop.com/tutorials/git-bare-vs-non-bare-repositories.html>

### 

### Initial configuration. Git settings.

To start working with git a developer must first tell git who he is. In the newly initialized working copy we have to set two values: user.name and user.email which will then be used by git to identify an author. Listing below shows how to do it.

$ cd ~/bob  
$ git config user.name "Bobby B. Bob"  
$ git config user.email "bob@gitcamp.com"

With git config command we *set* the values. To read the values later you can use the the similar syntax. Just omit the value, like this:

# reading current value of user.name  
$ git config user.name  
  
# reading the current value of email  
$ git config user.email

It is often useful to set these settings once for all your projects. Indeed, you are usually using just one name and email to contribute to the projects. Git supports it by using **global** settings.

Here’s how to set the same values globally:

$ git config --global user.name "Bobby B. Bob"  
$ git config --global user.email "bob@gitcamp.com"

Using the same principle it’s easy to *read* the global settings:

# read the global value of user.name  
$ git config --global user.name

Global settings act as a defaults. If git can’t find local value, it will go to read the global settings. Local settings override globals.

Here’s where the actual setting files are stored

# local settings  
$ cat .git/config  
  
# global settings  
$ cat ~/.config/git

Removing settings:

$ git config foo.bar "Baz"  
$ git config --unset foo.bar  
$ git config --remove-section foo  
  
Few more useful settings-related commands:

# show all global settings  
$ git config --global –list  
  
# open an editor with the settings file  
$ git config --global --edit

### Distributed nature of git - what is a remote

Git is a distributed system. Copies of the same repository can be stored in different servers. Even is all the servers are destroyed by a fireball storm, a single developers’ clone will help to restore the whole project history.

Such bare copies of a repository are known as “remotes”. For example, you might be working with a local copy that has two remotes: one is your work server that you connect to when you’re in the office and one is in your home private server.

This approach has more advantages than just reducing risk of data loss. We will how the distributed nature of git can be utilized in later sections of this tutorial.

By default, when you do git clone, you will get one remote called origin that points to the repository that you have cloned. Go to your project’s directory and execute:

$git remote –v  
origin /repos/camp.git (fetch)  
origin /repos/camp.git (push)

Once we’ll learn how to add extra remotes to the project this list will become longer.

The name origin is not special, it is just a default name that you can replace with your own when cloning a repository. Here’s how to do it:

$ git clone /repos/camp.git . **-o upstream**

The newly created repository will have one remote called upstream.

### First commit

Create a file and then add it to the repository. Learning the following commands:

$ echo "Horoscope App" > README.md

$ git status  
$ git add README.md  
$ git commit -m "Initial Commit"

# show the updated status  
$ git status

# show that the commit has been added   
$ git log  
$ git push -u origin master

### Sharing commit – git push

In this video show how to share the created code with the second developer and how to get the code from the other side. In this video second developer should add his own commit to prepare the ground for the next video.

In this section we also show and emphasize that the same command (git add) might do slightly different things depending on a context when it is used. Here, the command used to add the changed file to a commit.

### What is a commit, referring the commit

Explain that the commit is a full snapshot of a system at a given point in time. Show the git log command and explain that the commit is referred by the SHA-1 hash. Explain that the commit has a parent, a previous state of the system that this commit is based on.

### Getting the updates from the repository – git pull

Here the first developer should get the updates using git pull. We also summarize the first fully sequential workflow.

### Ignoring files

This section tells how to ignore the files with a help of gitignore.

$ mkdir node\_modules  
# empty folders cannot be added to git  
$ npm install colors  
$ git status  
  
# we don't need node\_modules in repository  
# .gitignore to the rescue  
$ vim .gitignore  
  
# Add entries:  
node\_modules  
.DS\_Store  
  
$ git status  
# node\_modules dissapear, but .gitignore is there  
  
$ git commit -am "Added gitignore"

Gitignore works per underlying folder hierarchy. If you have multi-environment project you might want to have multiple small gitignores per technology instead one file for everything.

### Deleting files, short-hand commit syntax

Explain the shorthand of the commit messaging. Tell that the files that are not tracked will not be committed with this syntax.

### I screwed up 1: fixing last commit

In this section we show what to do if the last commit didn’t quite go as you planned. Need to emphasize that this only works when the commit is not yet pushed.

## Section 2: Understanding git

In this secion we’ll start to explore the core features of git and understand a little bit better how it works.

### Git zones: working directory, index, pushed, stash

Speak about how file gets from one zone to the other. Emphasize that index contains the whole snapshot of the repository to be committed as the next commit. Show how to move the files from working to index and back.

### Basic collaboration

Showing the scenario when both developers changing code at the same time. Explain what happened on a picture: first show the extra commit in the log and then explain why it is there (merge commit).

### Git as graph

Show how git presents its data: a graph of commits, each commit pointing to its parents. Don’t talk about branches yet. Tell that commits are immutable – you can’t change existing commits even if flags like amend suggest that we’re changing it.

Tell that the power of git is that it can always return to one of this snapshots and restore system at that point in time.

### First conflict

When two developers are changing the same fragment of code at the same time it might cause conflict. For example, what to do if one developer updated the line and the other one has deleted it completely? Or they changed the same line of code.

Show how to resolve the conflict and create a merge commit.

### Ignoring files

This section explains how .gititgnore works.

### Why clone is called clone

Explain that when the repository is cloned, you get the whole history of a project. If the server crashes, each developer can restore it.

Mention that it is not always required. If you don’t need a whole history you can pass the --depth argument. This will create a shallow copy with only as much history as you need. Most frequently you pass --depth 1 to this command (if you don’t care about history at all):

$ git clone <url> --depth 1

This command creates a *shallow* repository (the repository that doesn’t have the full history). The shallow repository has a number of limitations but cloning it is much faster than a repository with the full history.

If you later decide that you still want that history back you can download it by calling

$ git pull --unshallow

Now as an example, let’s use our repository to create another bare clone. For this example I will create a true remote, that is located on another server.

# create another repository  
$ ssh [juriy@juriy.com](mailto:juriy@juriy.com)  
$ mkdir repos  
$ cd repos  
$ git init --bare camp.git

# getting back to my host  
$ git remote add central ssh://juriy@juriy.com/home/juriy/repos/camp.git  
$ git push --all central

Now we have added another remote to a project, another repository that we can use to push your code to or to collaborate with the teammates.

After pushing, we’ve delivered the whole history to the remote repository. This example showed that the migration from one server to the other, or restoring repository from one developers’ copy is really an easy task!

## Branching

We’ve all being taught in childhood to do one task at a time, however in a real production environments we often need to change our focus and work with many things at a time. It’s quite easy to imagine a scenario when one team is finishing the final touches before releasing version X of your app while the other team is adding features to a version X+1.

With the help of branches it is easy to manage these and many other scenarios when the development of a code is not linear.

### Basic work with branches

In this section we’ll show how to create a branch, swtich between the branches and add commits to a branch.

# show that there's one branch now  
$ git branch  
$ git checkout -b experiment  
$ git branch   
  
# modifying the code  
$ vim main.js  
$ git commit -am 'Adding experimental feature'  
  
# switch back to master, show that there's no changes there  
$ git checkout master

### Merging the code between the branches

Show how to pass the code between the branches

Git DAG  
We have looked already at how git stores its commits and now it is a good time to formalize our knowledge and add branches to the picture.

* Git stores the snapshots of the system that are called commits
* Commits track their parent commits
* Branch is merely a label on DAG
* If you commit while on branch, the tip of that branch will move, others will remain

### Git log – the graph version

Show how to add much more information to git log output. Show how to create alias.

# Let's look at the list of commit that we've got so far  
$ git log  
  
# will show huge output  
$ git log --pretty=oneline  
  
# much better but still very verbose  
$ git log --pretty=oneline --abbrev-commit  
  
# the output that we're looking for  
# Usually you're not interested in a whole history of a   
# project, so it makes sense to limit the output to a given number  
# of commits:  
$ git log --pretty=oneline --abbrev-commit -n 2

Add more explanation about how format works in git log, and how colors work

log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit --date=relative

### Gitk – a GUI tool to explore git log

### Deleting a branch

Show how to delete a branch that has not yet been published.

# to delete a local branch  
$ git branch -d experiment

### Publishing branches

Show how to publish the branch to a remote and get it from the other developer’s machine.

$ git push -u origin experiment  
  
# Or, to specify the different name, there's a more verbose syntax  
$ git push -u origin experiment:exp

Now, let’s get the same branch from the other developer’s repository. First we have to find out which branches are. By default git branch will only show local branches, so to see the branches from the server use the following command:

$ git branch -a

Now, when you know what are the names of remote branches, you’re ready to create remote tracking branches where you can commit your code.

# get updates from the server  
$ git pull  
  
# create the local version  
$ git checkout -b experiment origin/experiment  
  
# alternatively, this will also work  
$ git checkout experiment

### Kinds of branches

Local (working) branches

**Remote Tracking Branches** get their contents from the git clone, git pull and git fetch commands (git merge updates only the tracking branch).

**Remote Tracking Branches** should not be modified by users (don’t set your git branch to a remote tracking branch via git checkout and then try to modify the remote tracking branch).

**Remote Tracking Branches** get their contents from the remote repository.

**Tracking Branches** get their contents from the git clone, git pull and git merge commands (git fetch updates only the remote tracking branch).

**Tracking branches** can be modified by users.

After the user does a git commit in the local tracking branch, users can then use git push to publish changes from their tracking branch to the remote repository and remote tracking branches.

Show the whole workflow.

### Git fetch command

git fetch only updates remote tracking branches. Think about it as “download changes from server, but do nothing locally”.

### Understanding git pull.

Git pull is a two-phase process that is a shortcut to the following:

$ git fetch  
$ git merge

### Fast forwarding

Fast forward is a kind of merge when there’s no need to create a new commit, only the branch pointer has to be moved from one commit to the other. Fast forward is the safest possible kind of merge because it does not change the git graph, only moving the branch pointer to new places.

There’s a flag: --ff-only that tells merge or pull to do their job only if fast forward is possible:

$ git merge --ff-only origin/master  
$ git pull --ff-only

If fast forward is not possible, merge will not happen. In case of pull, the fetch phase will still occur.

### Deleting a remote branch

Deleting the branch that you have shared before with others requires slightly more commands than for the branch that is only yours. Indeed, you need to delete it from both local repository and remote one.

# to delete a local branch  
$ git branch -d bending-feature  
  
# to delete a remote branch  
$ git push origin --delete bending  
  
# others will have to run the following to remove the branch:  
$ git fetch origin --prune

### Optional: the default branch

Tell about master – the default git branch and how you could change master to some other name.

<http://stackoverflow.com/questions/4468322/switch-current-branch-in-git-bare-repository> - switching to the different default branch.

### Optional: How git stores branches

Show that git stores a branch as a simple reference to a commit.

$ cat .git/refs/heads/experiment

## Sharing work between branches

In this section we’ll talk about more advanced techniques to share the work between different branches:

* Rebase and its interactive mode
* Cherry Picking
* Using git log to see the commits between branches
* Using diff – seeing the difference
* Stashing uncommitted changes

## Navigating Git Graph

In this section we’ll learn how to navigate the git graph like a pro.

### Referrencing the commits

### Checkout and reset – overview

### Checkout

### What is HEAD. Reflog.

### Reset

### Rewriting public history

### Reverting unpublished commit

### Reverting published commit

### Squashing unpublished commits

### Undoing merge

### Undoing rebase

## Tags

### Creating a tag

Annotated vs Lightweight tag. Pushing tags to remote

### List all tags, getting code at tag

Use git log to list all tags

### Deleting a tag

## Git workflows

## Github

## GUI clients

## Integration and setup

* Hooks
* Git daemon
* Gitlab

## Other topics

### Line endings in git

Show in practice how different line ending settings will work.

### Why shared repositories should be bare

Good article on bare/non-bare repositories: <http://bitflop.com/tutorials/git-bare-vs-non-bare-repositories.html>

<http://stackoverflow.com/questions/7632454/how-do-you-use-git-bare-init-repository>

### Merge two git repositories (projects) into one

Tell how to merge the projects and that the project might have more than one initial commit.

## Notes

Comparing different version control systems should be better done in a later sections, because some listeners might not have a notion of what is "commit" just yet.

## I screwed up!!! Ideas.

1. Committed with the wrong message

2. I was committing to the wrong branch

3. I forgot to make a branch an committed to the master

4. After rebase the code is a mess

5. After git-pull the code is a mess