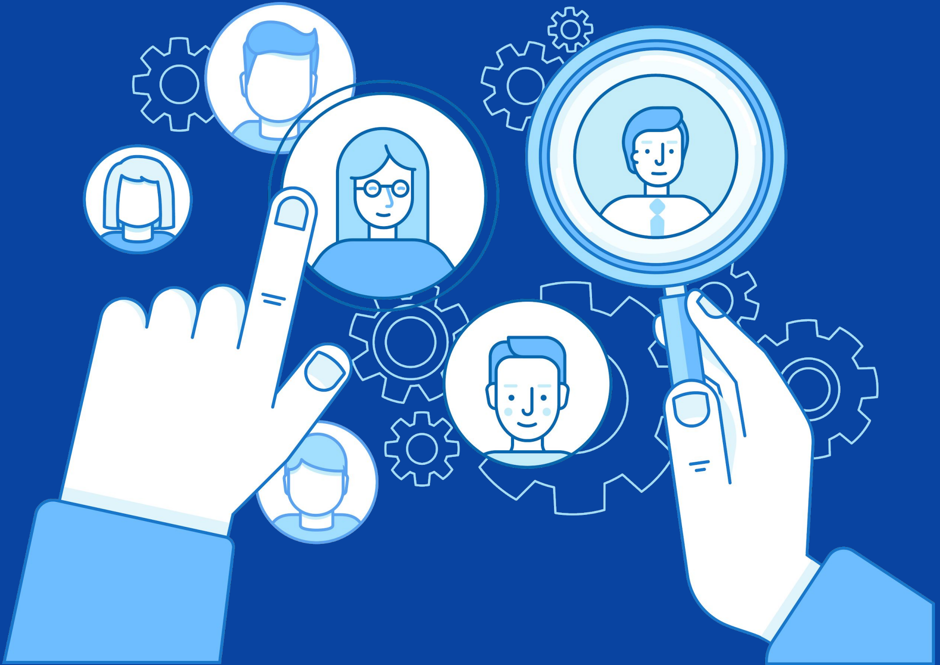
Digital Career Institute

Versioning and Collaboration



Digital Career Institute

Goal of the Module

By the end of this sub module, you will

- Understand common use cases for version control software
- Create local repositories using the terminal
- Create remote repositories on GitHub
- Understand the basic git workflow
- Use commits and branches to create version histories for projects

Digital Career Institute

Topics

- Introduction to Version Control Systems
  - Including a brief history
- The git program
  - Distributed version control
- Git core concepts and commands
  - Local and remote repositories
  - Staging
  - Basic commands and workflow

Digital Career Institute

Introduction to Version Control Systems

Digital Career Institute

Version Control Systems

Very quickly when developing software you will notice the need for versioning your work. There are many reasons why this is the case.

Many things are very hard to achieve without a formal version control system in place, such as

- Multi-person teams working on one project
- Maintaining old versions and developing new ones
- Fixing problems in multiple versions
- Maintaining multiple versions before release
    - one version for testers
    - one version for developers
    - one for marketing

Mr Cyborg?

Imagine you are developing a mobile OS “Cyborg” 🤖.

You release 1.0, then 2.0. You start 3.0 and find a bug in 1.0 & 2.0!

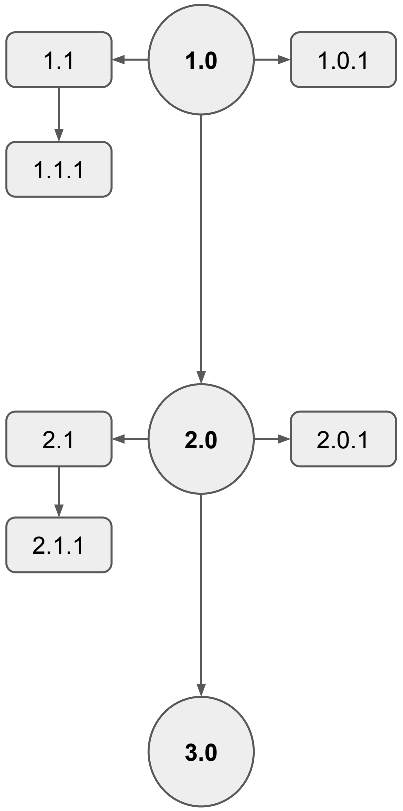
Some users have 1.0 (old phones can’t use 2.0) so fixes are made: 1.1, 2.1 and 3.0!

Then a critical security bug is discovered affecting all 1.0, 1.1, 2.0, 2.1 and 3.0.

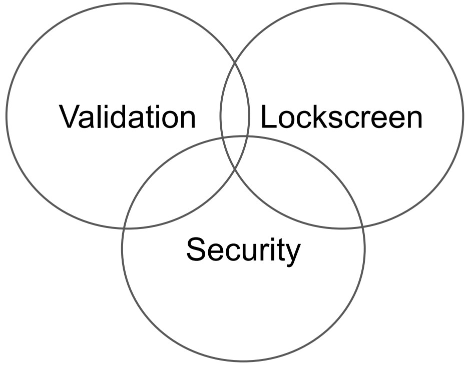
The fix is the same for all versions, and you need an emergency fix for all versions, even ones without the previous fix.
So now you have 1.0.1, 1.1.1, 2.0.1, 2.1.1 and 3.0.1 😨

Jumping between versions and maintaining them is very real in development!

Clearly, a system is needed.



Keep calm and collaborate



Another scenario for the Cyborg OS!

What if there are multiple developers?

Jane is working on login validation.
Alex is working on the lockscreen.
Mehmed is working on data security.

Here we have three versions being worked on, often dealing with overlapping content.

When Mehmed finishes his work, Jane and Alex must update their working versions, merging the work from Mehmed to their versions while continuing the work they have already done.

Now imagine this when there are hundreds of developers working on a project!

Clearly, a system is needed.

A system

These are some of the issues solved by a VCS (Version Control System) - also known as

- revision control
- source control
- source code management

These systems are often mainly used for source code, but they can also be used for almost any versioning

- Documents - like markdown files
- Documentation for projects
- Data files - like language translation files
- Configuration files

VCS history

The history of version control goes back to the ‘60s and ‘70s.

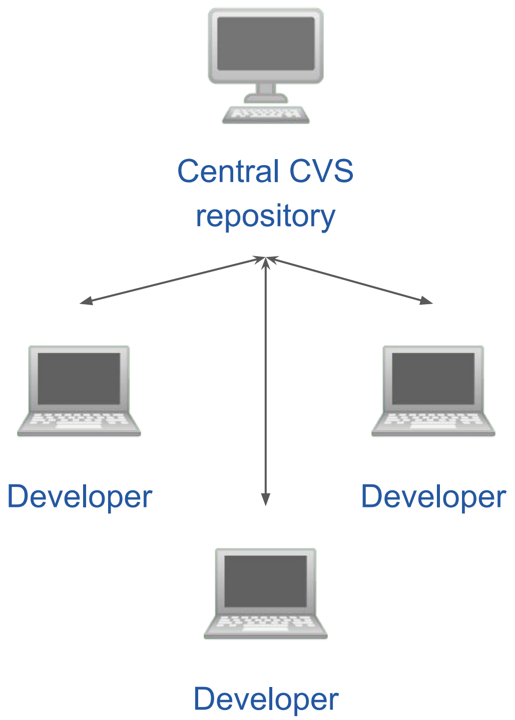
One of the first widespread version control systems was CVS (Concurrent Versions System), released in 1986.

CVS introduced a client-server model that became the standard for version control for years.

In this model, a central server stores the files and history data in a central *repository*.

Programmers acquire or “*check out*” copies of the repository (or parts of it) to their computer (the client).

After working, programmers “*check in*” their changes.



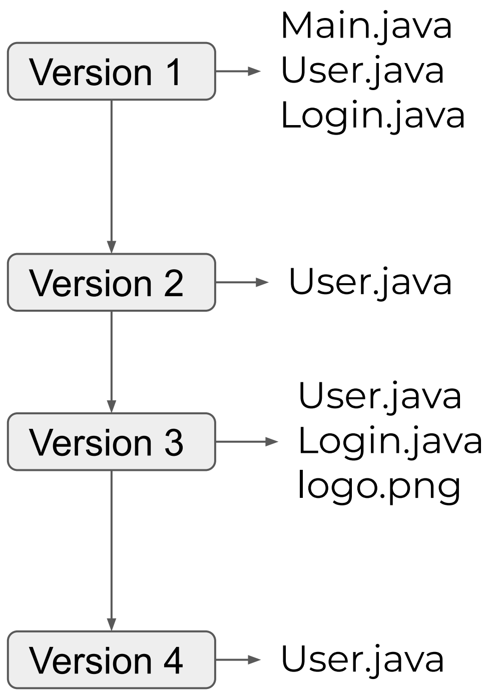
VCS to SVN ⚛️

CVS used a system where modifications were tracked on a file-by-file basis: a part of your changes might fail and a part of them work.
This also means you tracked the changes (histories) of individual files, instead of tracking sets of changes.

The next generation of version control was defined by **SVN** (Subversion).

This new generation introduced “atomic commits”, meaning your set of changes was tracked all together instead of file-by-file. One set of changes is called a “**commit**”

You could track the history of your entire project from change to change and if one change conflicted, your entire set of changes would be halted.



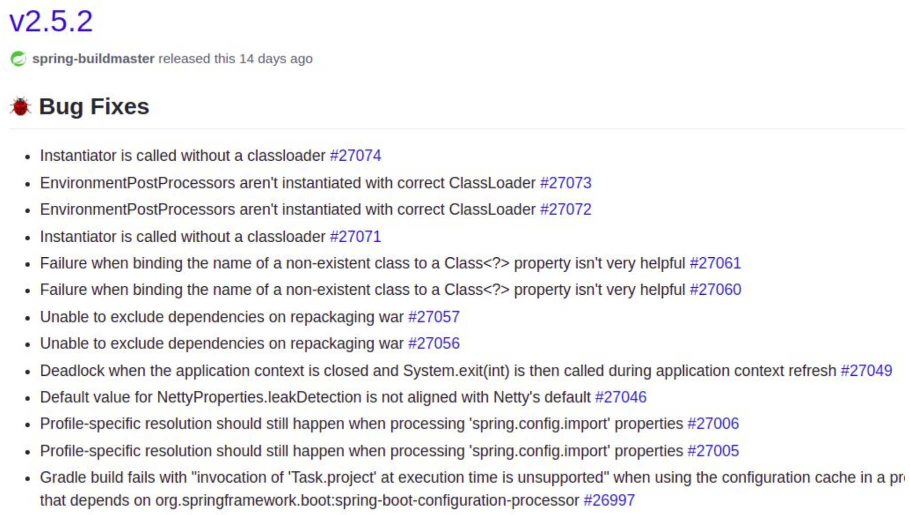
Subversion

Along with a set of changes (a commit), developers add their own messages (a commit message).

This meant that it is very easy to generate a list of changes between versions by looking at the messages!

These messages can be gathered together between major versions and edited to form a changelog (list of changes) between versions!

This is why often there are specific rules as to what those messages should look like.



v2.5.2

spring-buildmaster released this 14 days ago

Bug Fixes

- Instantiator is called without a classloader #27074
- EnvironmentPostProcessors aren't instantiated with correct ClassLoader #27073
- EnvironmentPostProcessors aren't instantiated with correct ClassLoader #27072
- Instantiator is called without a classloader #27071
- Failure when binding the name of a non-existent class to a Class<?> property isn't very helpful #27061
- Failure when binding the name of a non-existent class to a Class<?> property isn't very helpful #27060
- Unable to exclude dependencies on repackaging war #27057
- Unable to exclude dependencies on repackaging war #27056
- Deadlock when the application context is closed and System.exit(int) is then called during application context refresh #27049
- Default value for NettyProperties.leakDetection is not aligned with Netty's default #27046
- Profile-specific resolution should still happen when processing 'spring.config.import' properties #27006
- Profile-specific resolution should still happen when processing 'spring.config.import' properties #27005
- Gradle build fails with "invocation of 'Task.project' at execution time is unsupported" when using the configuration cache in a pr that depends on org.springframework.boot:spring-boot-configuration-processor #26997

Version history

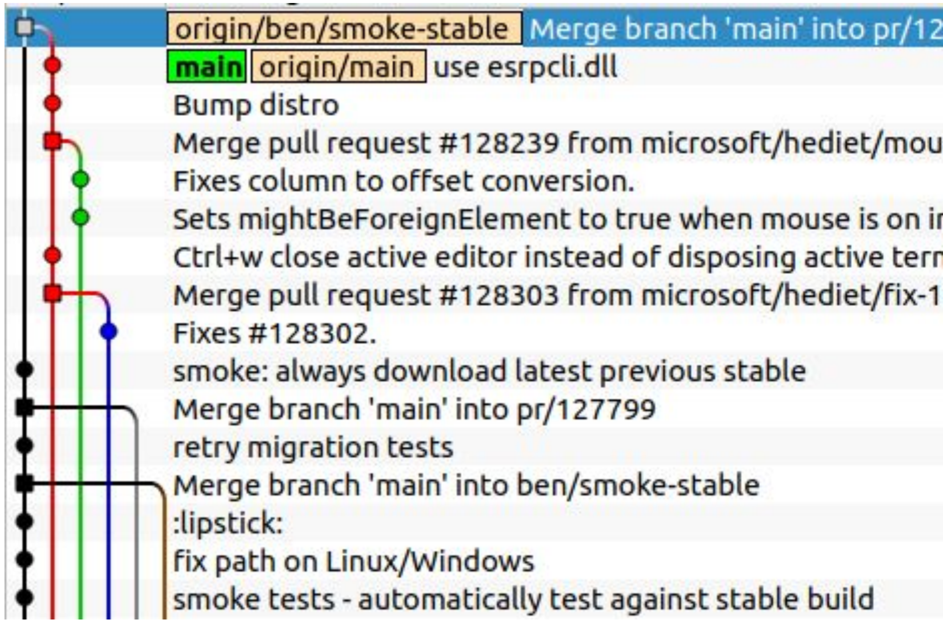
So you have sets of changes and you maintain multiple versions.

To tell different versions of an application apart, version control systems use a system called **branching**, like branches on a tree.

Branches can contain different commits!

Branches can be created and later removed - some of them might represent a specific release version and some might represent features that are still being worked on.

Usually there is one Main version, where other branches originate from.



```
origin/ben/smoke-stable Merge branch 'main' into pr/12
main origin/main use esrpcli.dll
Bump distro
Merge pull request #128239 from microsoft/hediet/mou
Fixes column to offset conversion.
Sets mightBeForeignElement to true when mouse is on i
Ctrl+w close active editor instead of disposing active term
Merge pull request #128303 from microsoft/hediet/fix-1
Fixes #128302.
smoke: always download latest previous stable
Merge branch 'main' into pr/127799
retry migration tests
Merge branch 'main' into ben/smoke-stable
:lipstick:
fix path on Linux/Windows
smoke tests - automatically test against stable build
```

At the core of the lesson

- Version Control Systems are necessary
- They help with many issues, such as
  - Maintenance of multiple releases
  - Collaboration
  - Tracking what has changed between versions
- There are many VCS programs out there
- CVS was one of the first ones
- CVS was superseded by Subversion (SVN)
- Modern version control systems
  - Track sets of changes over time
  - One set of changes can have changes to many files
  - Each changeset has a comment describing it
  - Changesets are made on a specific branch
  - Branches are parallel versions of your application

Digital Career Institute

GIT

git

If Subversion was the previous generation, then **git** is the current one.

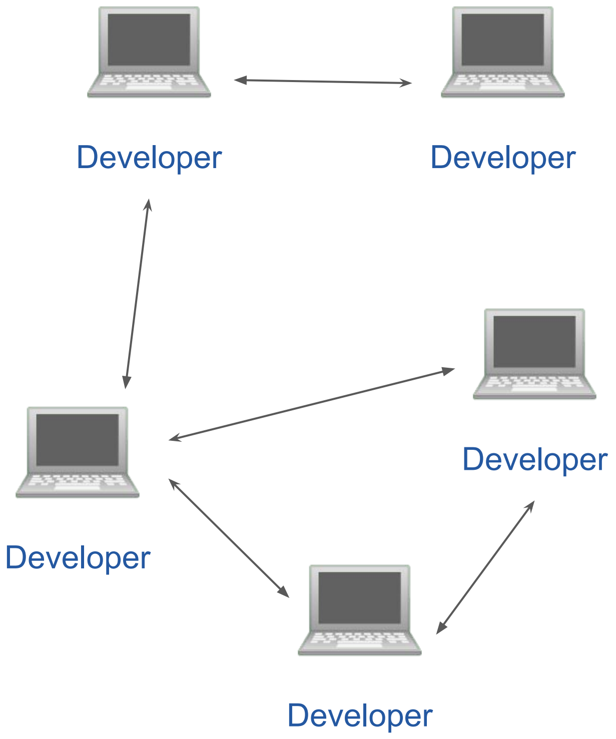
Git was created by Linus Torvalds in 2005 for development of the Linux kernel.

Many things are similar for git than historical VCSs, but it is a new generation of version control.

One fundamental change is that git is a distributed version control system (DVCS).

All repositories are technically equals.

In the traditional model there were servers and clients - in distributed systems you just have servers.



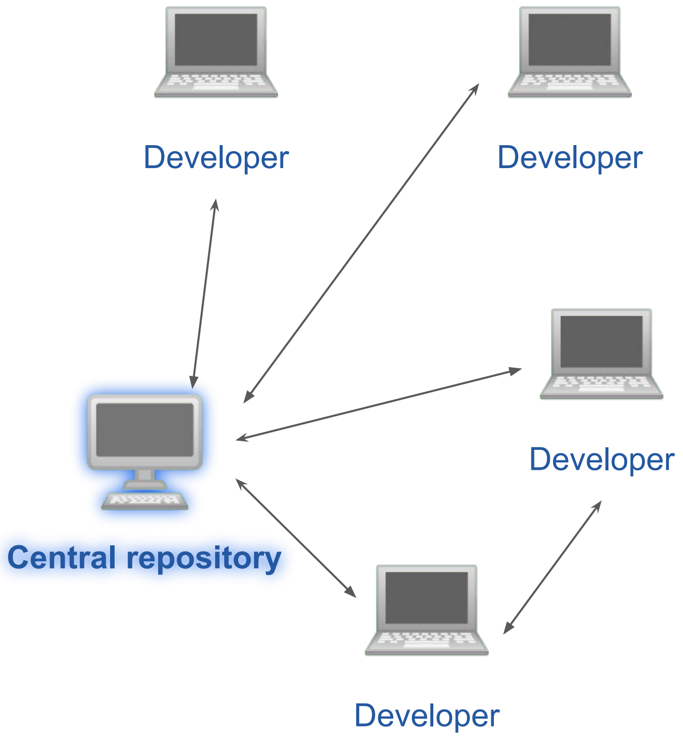
git

In distributed models, you can transfer work directly between developers. Still, there often is a central repository that is the main source of truth.

Central repositories can be internal services for companies, or they can be services provided by other service providers.

There are many git repository hosting services

- GitHub
- GitLab
- Bitbucket
- SourceForge



git

- Decentralized / Distributed
- Free and open source
- Very fast
- Scales well for **very** large project, like these public projects:
    - Linux 🐧
    - Visual Studio Code
    - React
    - Homebrew
- Rapid branching
    - Making a new branch is easy, fast and lightweight
- Used in practically all modern software development
- Large ecosystem of tools and services
    - Hosting services
    - Visualization tools
    - Integrations to development tools

Git CLI

There are many tools for using git, the main one is still the git CLI (Command Line Interface).

Meaning that a lot of git usage is done purely in the terminal, using the **git** program.

After installing git, you have to know a few central commands and concepts to work with it.

1. Any directory (folder) can be made into a git repository
2. Repositories can also be “cloned” from an existing source
3. You can have repositories inside repositories, but probably should not

Always stay aware of what directory you are working in. Usually it is best to have a system, at a minimalistic approach is to have one directory where you keep all your git repositories.

For example ~/projects/

It’s best to start a system immediately and follow it at all times to avoid confusion ❗

Git configuration

Digital Career Institute

The git program can and should be configured with your custom information

These configurations are saved in your home directory, inside the hidden **.gitconfig** file

If this file does not exist, you have not configured git 🤷

You can edit the file directly or you can configure git using the command **git configure**

At the very least you should have your email and your name configured, this way git knows who you are. Your email can also help you work with external repositories.

```
git config --global user.name "Joel Peltonen"
git config --global user.email "joel.peltonen@example.com"
```

Repository here, repository there…

To create a repository out of a regular directory, you can use the **git init** command.

Running **git init** will transform your working directory into a git repository.

This command creates the repository by making a hidden directory called ***.git*** inside the working directory.
This means that to “unmake” your repository by just deleting the .git directory!

Remember that repositories can be inside repositories?
⚠️ It is easy to accidentally run *git init* when working in a subdirectory of an old project
⚠️ It is also possible to accidentally run git init in the parent directory of a project

So be mindful of what you are doing!
If this happens, just undo it by deleting the invalid .git directories.
But make sure you delete the right one!

Repo here, repo there...

Digital Career Institute

The .git directory contains many files that form the core of your git repository:

- History of all commits in the repository
- Information about connections to other repositories
- All information about branches, like which is the currently active one

Feel free look around and view the contents, but don't edit the files!

The contents of the files should only be edited by using commands found in the main git program. Git is very sensitive to changes and it is very easy accidentally break something.

By the way, *repository* is such a long word... usually we just call them "repo" for short.

Stay sharp!

If you have multiple directories in your repository, the .git directory will only be found in the top level of your repository, even though the other directories are part of the project - yet another reason to keep vigilant what is the working directory!

A directory listing for your project might look like this for example - note only one .git directory

```
projects/
    my-application/
        .git/
        src/
            java/
            static/
        docs/
        libs/
```

At the core of the lesson

- Git is a version control system
- Git is used with the **git** command
- Git is a distributed system…
- … but often there is some central repository

- Any folder can be a repository
- **git init** is the command to create a repository
- Repositories are called repo for short

- Usually you configure your identity
- So git knows who made those changes!

Digital Career Institute

Basic git commands & workflow

Digital Career Institute

Git commands

So you use git with the command line **git** program.

❓How do you *actually* do something with it❓

We will learn key commands for using git right now

Before we start, know that git is really only interested in files.

- If you create a file, git is interested in that file.
- If you create an empty directory, git is not interested in it.
- If you create a file inside an empty directory - *oh boy* - git is interested in it!

Your approach / method / strategy of using git is called a workflow

Workflows can differ from company to company
Workflows can differ from project to project
Workflows can differ from person to person
Choose one and stick with it!

Git commands: git status 1

How do you find out what git thinks about a directory?

This is (*according to the author of these slides*) the most important git command.

```
git status
```

Run that command anywhere.
Run that command all the time.

That command tells you

- Are you in a git repository
- If yes, what is the status of that repository
    - Are there changes to the files
    - What branch are you currently on

Git commands: git status 2

```
dci@dci-laptop:~/projects/not-awesome-project $ git status
fatal: not a git repository (or any of the parent directories): .git
```

____
_

```
dci@dci-laptop:~/projects/my-awesome-project $ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

Git commands: git status 3

```
dci@dci-laptop:~/projects/my-awesome-project $ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   README.adoc

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    newfile

no changes added to commit (use "git add" and/or "git commit -a")
```

Git commands: git add 1

You created a file and edited another one? Nice!

Keep making changes as long as you want!

You then need to instruct git about the changes. You do this by **staging** those changes.

Staging means you tell git which changes you want it to include in your next commit. You do this with the **git add <path>** command.

The changed files are then added to an imaginary staging area:

```
dci@dci-laptop:~/projects/my-awesome-project$ git add README.adoc
dci@dci-laptop:~/projects/my-awesome-project$ git add newfile
dci@dci-laptop:~/projects/my-awesome-project$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
	modified:   README.adoc
	new file:   newfile
```

Git commands: git add 2

The *git add* command accepts any path as the third argument, so you can add entire directories into the staging area by providing a path to a directory.

Remember that **.** is the shortcut to the current working folder. This means you can run the command **git add .** to add all of the changes within the current directory to the staging area.

```
dci@dci-laptop:~/projects/my-awesome-project$ git add .
dci@dci-laptop:~/projects/my-awesome-project$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
	modified:   README.adoc
	new file:   newfile
```

Git commands: git commit 1

Now your changes are staged and you can finalize your commit!

You do this with the **git commit** command.

Remember that commits have a commit message attached to them?

If you run git commit directly, it will open up a text editor and ask you to write a message for your commit. You can configure what text editor git uses later, if you need to.

You can provide a message directly on the command line using the **-m** flag and then providing your commit message in quotation marks. These are sometimes called *inline* commit messages and they are limited to one line of text.

```
dci@dci-laptop:~/projects/my-awesome-project$ git commit -m “Add demo feature”
[main 7e1cf288f0] Add demo feature
 2 files changed, 1 deletion(-)
 create mode 100644 newfile
```

Git commands: git commit 2

You made a commit!

Your commit now exists in your local repository, nobody else has it.

You can create multiple commits when working - often projects or companies have some guidelines or policies when to make commits and what makes a good commit.

Examples of commit guidelines

- commit only working code
- commit at the end of every day, even if your code is not working
- commit only when you have a logical set of changes

Examples of commit message guidelines

- limit the subject line to 50 characters
- use complete sentences
- always start your message with an issue ID
- write your commit message in imperative (“Fix bug” not “Fixed bug”)

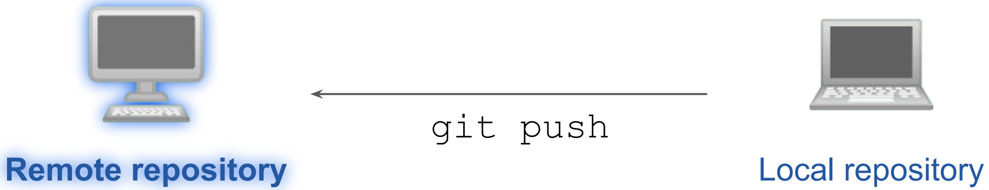
Git commands: git push

Making a commit stores your staged changes as one commit, in the history of your current branch, in of your local repository. To send those changes to somewhere, you need to

git push

Doing a push means you are pushing the commit(s) that you have into some remote repository. This does require that your repository is connected to some other repository!

Git also compares the repositories together; does the remote have changes you don't have? Are there conflicts where changes are made to the same files?



Git commands: git pull

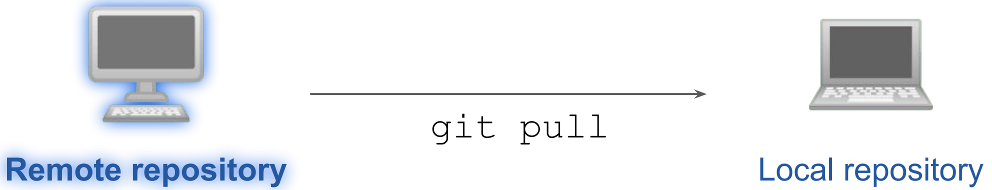
If there are changes in the remote or central repository that you don't yet have in your local repository, you can do the opposite of pushing.

git pull

This updates your local repository with everything that the remote knows.

It is advisable to run git pull often, if you can!

It is possible that there is a conflict if you are locally working on a file that someone else has changed in the remote repository - and it's better to notice the conflict as soon as possible!



Git commands: git log

To read the commit history of what you are working on, you can run the command **git log**.
This command can be *very* flexible, we will focus on the basics.

Notice how each commit has a commit id, an author, a date and a message!
To exit out of the log, press **q**.

```
dci@dci-laptop:~/projects/my-awesome-project$ git log
commit 7e1cf288f03e8481b77d9505d9098a994b2bce9e (HEAD -> main)
Author: Joel Peltonen <joel.peltonen@digitalcareerinstitute.org>
Date:   Fri Jul 9 17:45:11 2021 +0200

    Add demo feature

commit 7a1c923fecacd4abafda82fa8c2fc6be3bc4e761 (origin/main, origin/HEAD)
Merge: 0b604f5e3b 3de58c2340
Author: Andy Example <example@example.org>
Date:   Fri Jul 9 14:18:18 2021 +0100

    Merge branch '2.5.x'
```

At the core of the lesson

git status	- what is happening
git add	- add changes to staging area
git commit	- commit staged changes
git push	- push commits to remote repo
git pull	- pull commits from remote repo
git log	- view log of changes

Digital Career Institute

Branching

Digital Career Institute

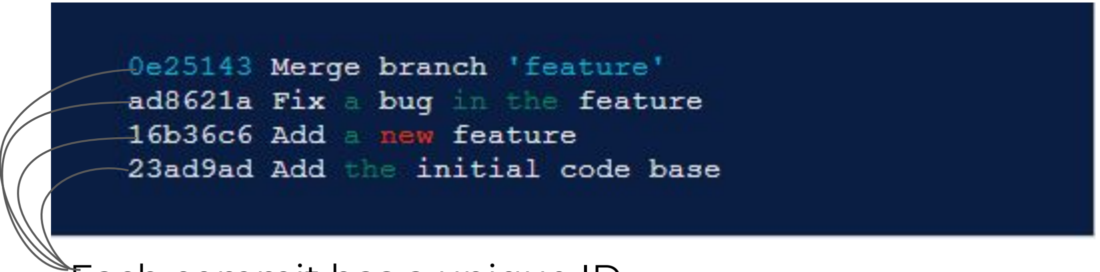
Git checkout

After creating commits on your git repository, you can see the commits using:

```
$ git log
```

Or for less details:

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



```
0e25143 Merge branch 'feature'
ad8621a Fix a bug in the feature
16b36c6 Add a new feature
23ad9ad Add the initial code base
```

Each commit has a unique ID

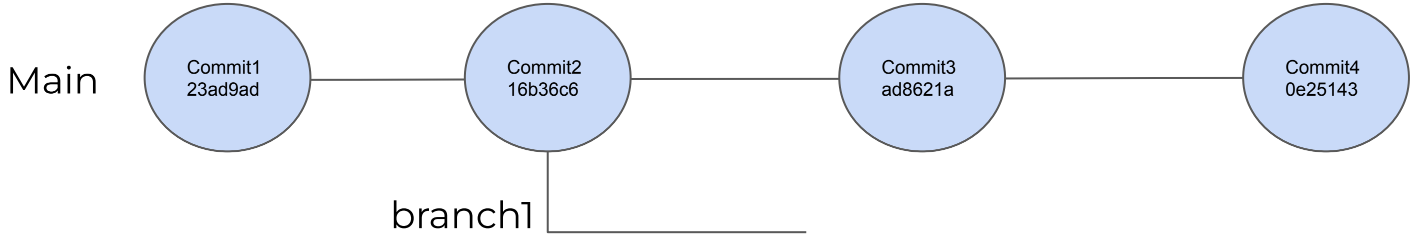
To move through the history:

```
$ git checkout <commit>
```

Example :

```
$ git checkout  ad8621a
```

Git branch



Create new branch called branch1 start from commit2

git checkout -b <branch name> [start point]

```
$ git checkout -b branch1 16b36c6
```

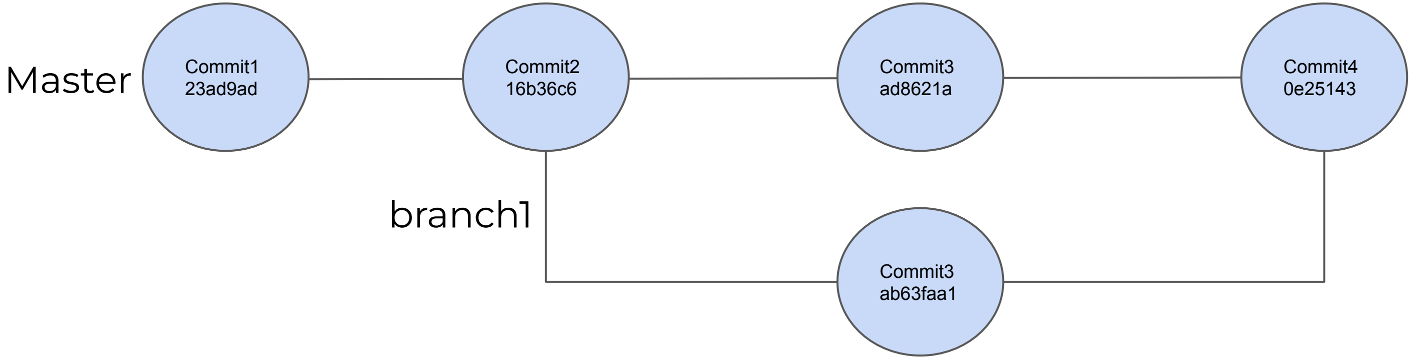
Git branch

Digital Career Institute

Inspect all existing branches:

```
$ git branch -a -v
```

Git merge



```
$ git merge branch1
```

Merge Strategies:

- Fast Forward
- Recursive
- Ours
- Octopus
- Resolve
- Subtree

Git remote & clone

Shows all remotes:

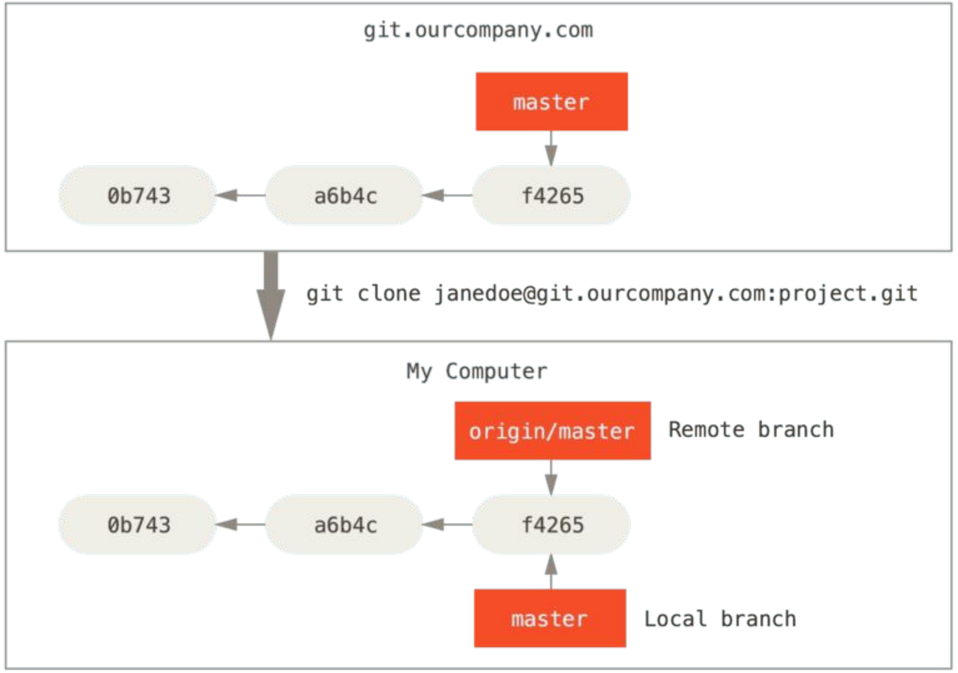
```
$ git remote
```

Add a remote:

```
$ git remote add <shortname> <url>
```

Clone an existing repository

```
$ git clone <url>
```



```
git clone janedoe@git.ourcompany.com:project.git
```

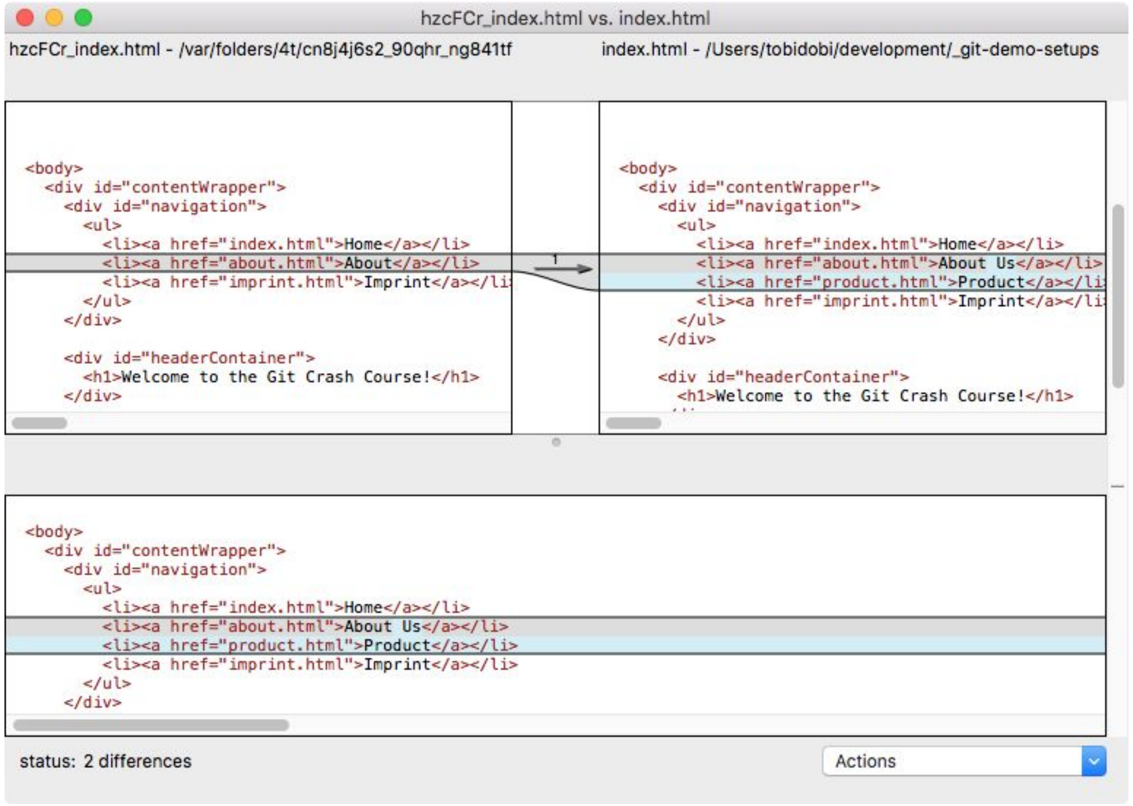
Merge conflicts & PRs

Digital Career Institute

Git merge conflicts

Depends on the tool configured:

```
$ git mergetool
```



hzcFCr_index.html vs. index.html

hzcFCr_index.html - /var/folders/4t/cn8j4j6s2_90qhr_ng841tf

```
<body>
  <div id="contentWrapper">
    <div id="navigation">
      <ul>
        <li><a href="index.html">Home</a></li>
        <li><a href="about.html">About</a></li>
        <li><a href="imprint.html">Imprint</a></li
      </ul>
    </div>

    <div id="headerContainer">
      <h1>Welcome to the Git Crash Course!</h1>
    </div>
```

index.html - /Users/tobidobi/development/_git-demo-setups

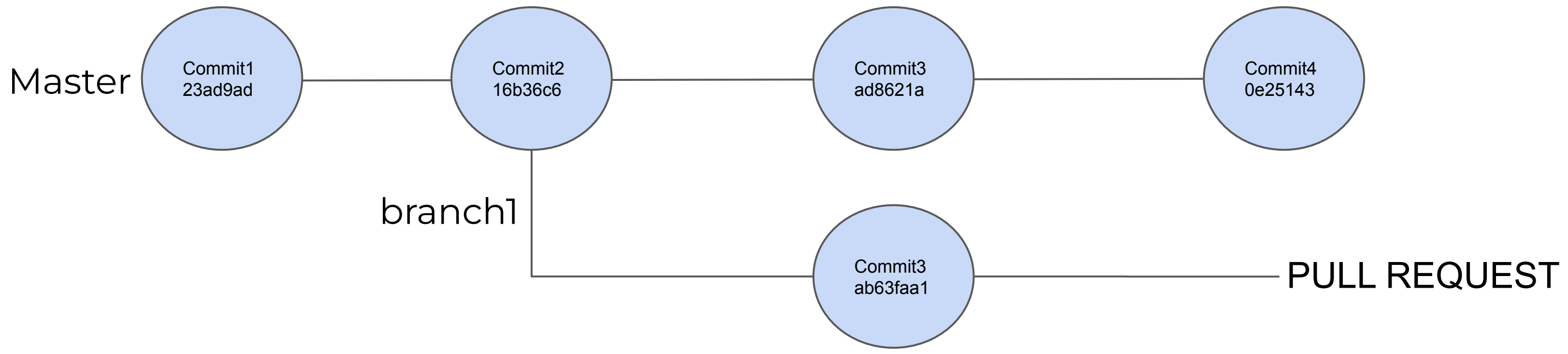
```
<body>
  <div id="contentWrapper">
    <div id="navigation">
      <ul>
        <li><a href="index.html">Home</a></li>
        <li><a href="about.html">About Us</a></li>
        <li><a href="product.html">Product</a></li
        <li><a href="imprint.html">Imprint</a></li
      </ul>
    </div>

    <div id="headerContainer">
      <h1>Welcome to the Git Crash Course!</h1>
```

```
<body>
  <div id="contentWrapper">
    <div id="navigation">
      <ul>
        <li><a href="index.html">Home</a></li>
        <li><a href="about.html">About Us</a></li>
        <li><a href="product.html">Product</a></li>
        <li><a href="imprint.html">Imprint</a></li>
      </ul>
    </div>
```

status: 2 differences

Git pull requests

Digital Career Institute



Git pull requests

1. Creating a Fork
2. Keeping Your Fork Up to Date
3. Doing Your Work
4. Submitting a Pull Request
5. Accepting and Merging a Pull Request

Git pull requests

Digital Career Institute

Benefits:

- **Peer Review**
- **Sufficient testing and better stability**
- **Reducing conflicts**
- **Continuous Delivery**
- **Clearer responsibility**

Documentation

Digital Career Institute

Documentation

1. Git Handbook
2. Git Flow
3. Git Glossary
4. Git Documentation
5. Git Developer Beginner
6. Git Config Cheatsheet
7. Git Cheatsheet
8. Prepare to use GitHub
9. Push & Pull (video, 1 min)
10. Working with Remotes

THANK YOU

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute
DCI