Version Control System

with Git

Technical University -- Propaedeutic -- June 2016

Why should I use a version control system?

Why should I use a version control system?

Collaboration

Storing Versions

Restoring Previous Versions

Understanding What Happened

Backup

https://www.git-tower.com/learn/git/ebook/en/mac/basics/why-use-version-control

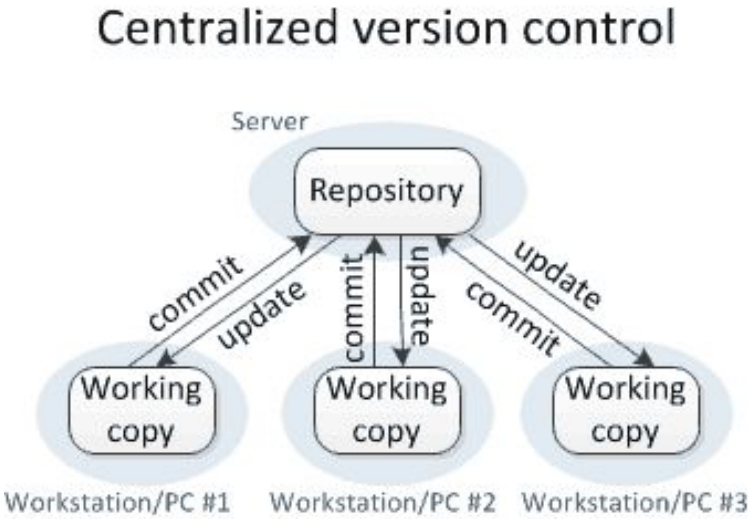
Agenda

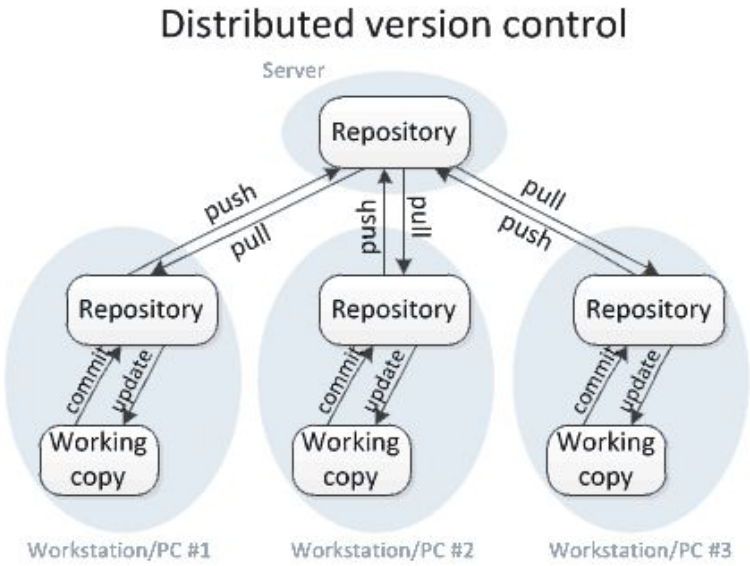
- Why should I use a version control system?
- Terms
- Centralized vs. Distributed version control
- Workflow & File Status Lifecycle in Git
- Use Git (practical)
  - Configuration
  - Initialisation & Clone Repository
  - Status
  - Stage/Commit
  - Log
  - Remote Repository
  - Branch
  - Stash
  - Revert & Reset
  - Conflicts
- Git workflow for software projects
- Git rules in collaboration

Terms

- Repository (local, remote)
  - A concept that refers to a data structure
- Files
  - All kinds of data e.g. text, pdf, images, audio etc.
- User
  - People who modify the files in the repository
- Log
  - Records of activities which are performed by the user
- Command Line
  - CL is used to interact with Git
- Workflow
  - Is a repeatable pattern of activities

Centralized vs. Distributed version control

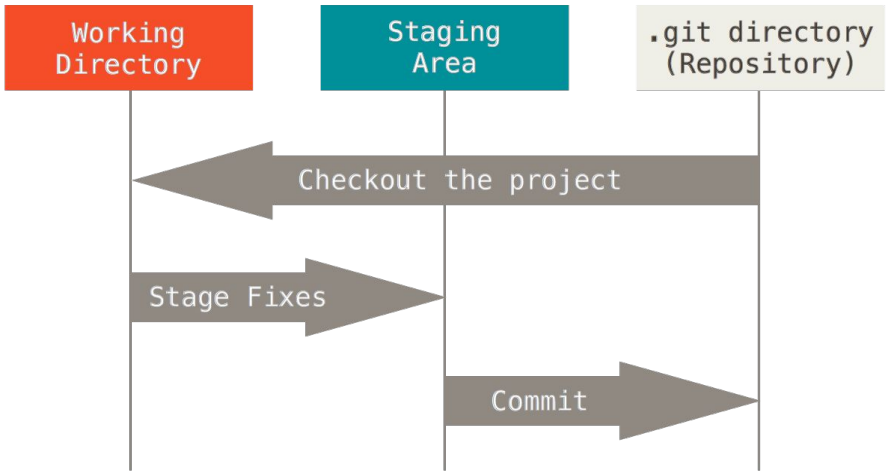




https://homes.cs.washington.edu/~mernst/advice/version-control-fig3.png

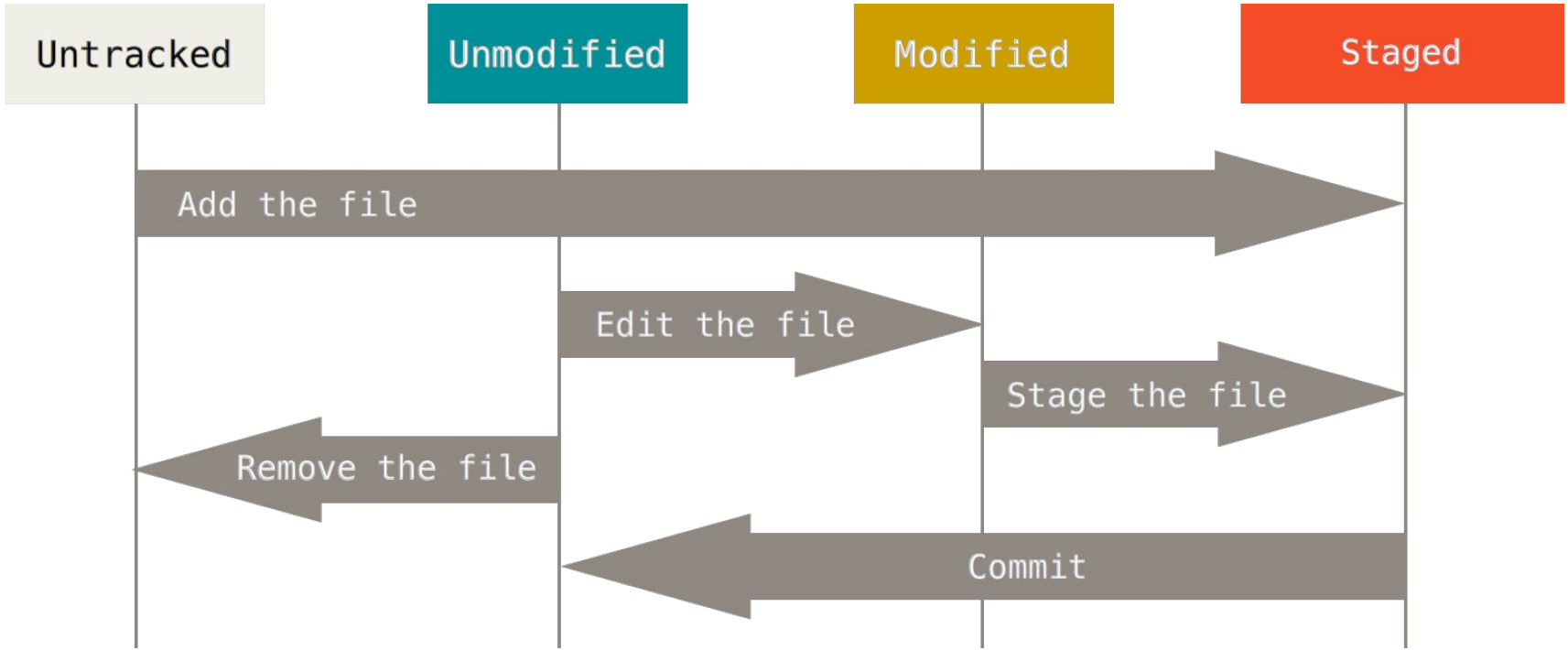
Git Basic Workflow

1. You **modify** files in your **working directory**.
2. You **stage** the files, **adding snapshots** of them to your staging area.
3. You do a **commit**, which takes the files as they are in the **staging area** and stores that snapshot **permanently** to your Git directory.



https://git-scm.com/book/en/v2/book/01-introduction/images/areas.png

File Status Lifecycle



https://git-scm.com/book/en/v2/book/02-git-basics/images/lifecycle.png

Usage Git

First steps ...

Configuration

Use git config to set or change attributes.

The settings are related to either for the global system or for an individual project.

Name and e-Mail address is an information which are used for git commits.

```
git config --global user.name "Burak Cetinkaya"
git config --global user.email cetinkaya@tu-berlin.de
```

List settings

```
git config --list
```

Initialization & Clone Repository

You can get a Git project using two main approches:

- Take an existing project or directory and imports it into Git
  - `git init`
  - `git remote add origin git@gitlab.tubit.tu-berlin.de:username/gitfolder.git`

- Clone an existing Git repository from another server.
  - `git clone username@host:/path/to/repository`

Status

Displays the status of the working directory.

Differences between the working directory and the Staging area. Informs you which files are:

- staged
- tracked
- untracked
- modified.

No information about commits.

```
git status
git status -s
```

Stage and Commit

Push file(s) to the staging area

```
git add <filename>
git add *
```

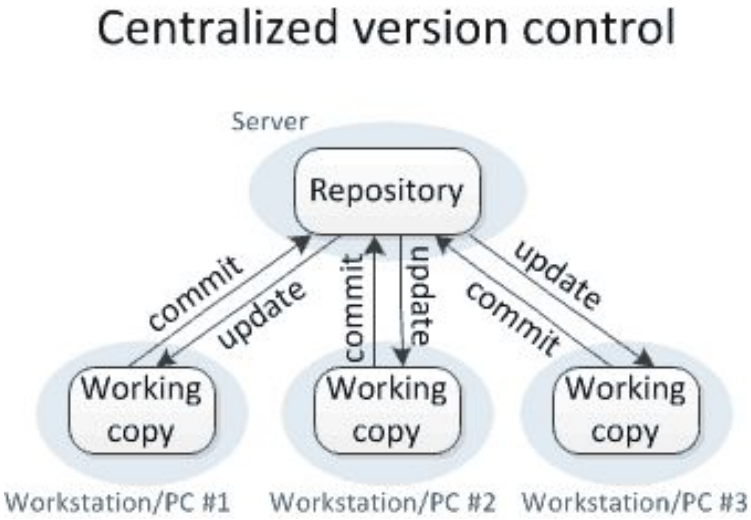
Commit the changes to the history in .git folder

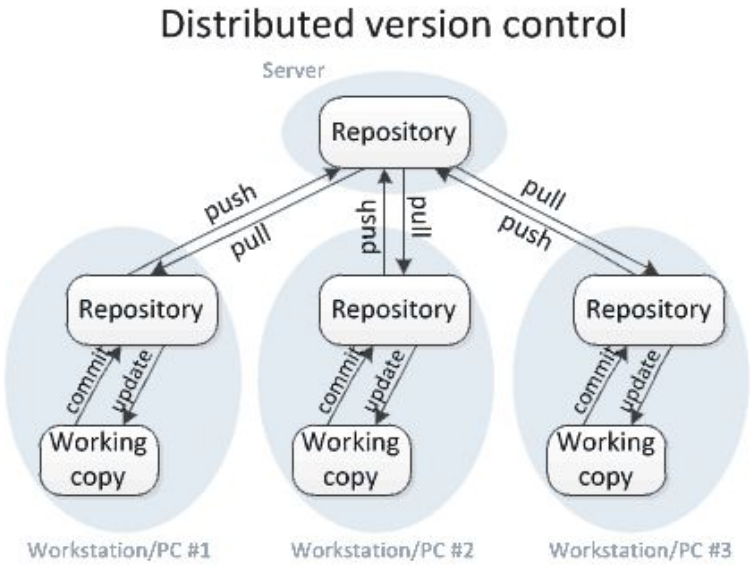
```
git commit -m “Commit-Message”
```

Agenda

- Short repetition
- Exercise
  - SSH Key
  - Add users into the project
- Branches

Centralized vs. Distributed version control

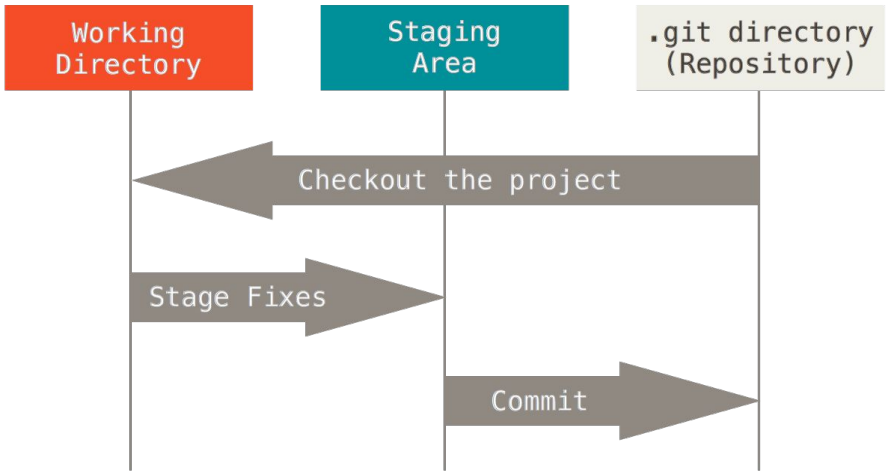




https://homes.cs.washington.edu/~mernst/advice/version-control-fig3.png

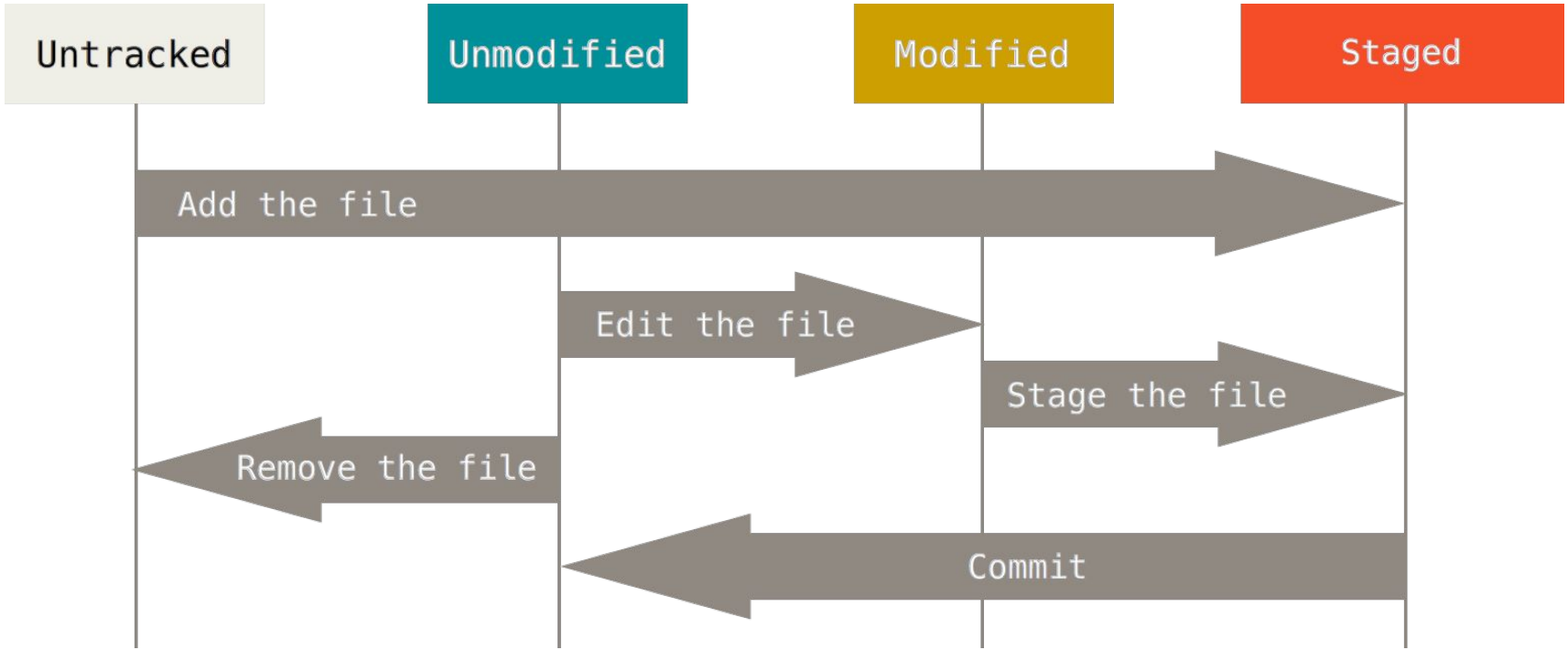
Git Basic Workflow -> 3-Steps

1. You **modify** files in your **working directory**.
2. You **stage** the files, **adding snapshots** of them to your staging area.
3. You do a **commit**, which takes the files as they are in the **staging area** and stores that snapshot **permanently** to your Git directory.



https://git-scm.com/book/en/v2/book/01-introduction/images/areas.png

File Status Lifecycle

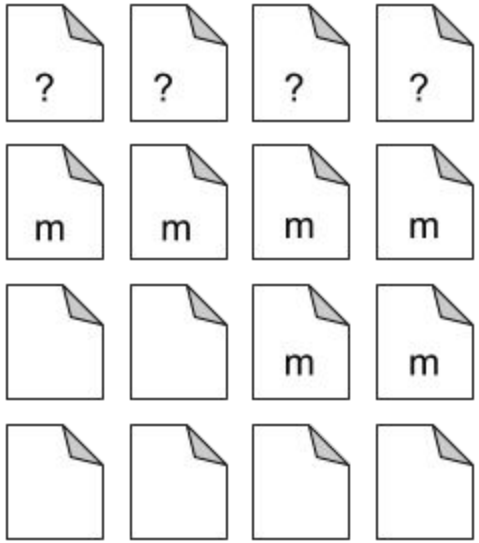


https://git-scm.com/book/en/v2/book/02-git-basics/images/lifecycle.png

Working Directory
Real Files

Staging Area
Index

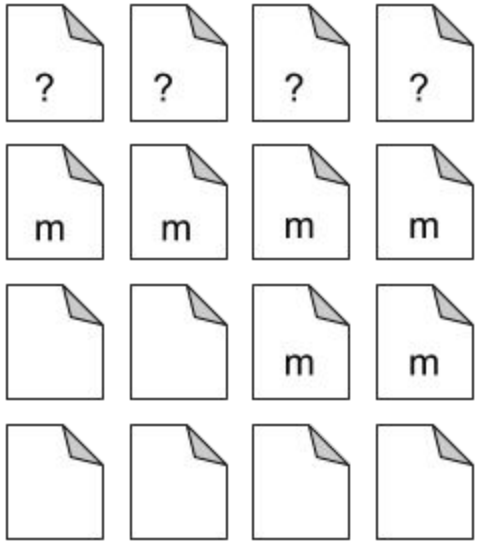
.git Directory
Local Repo



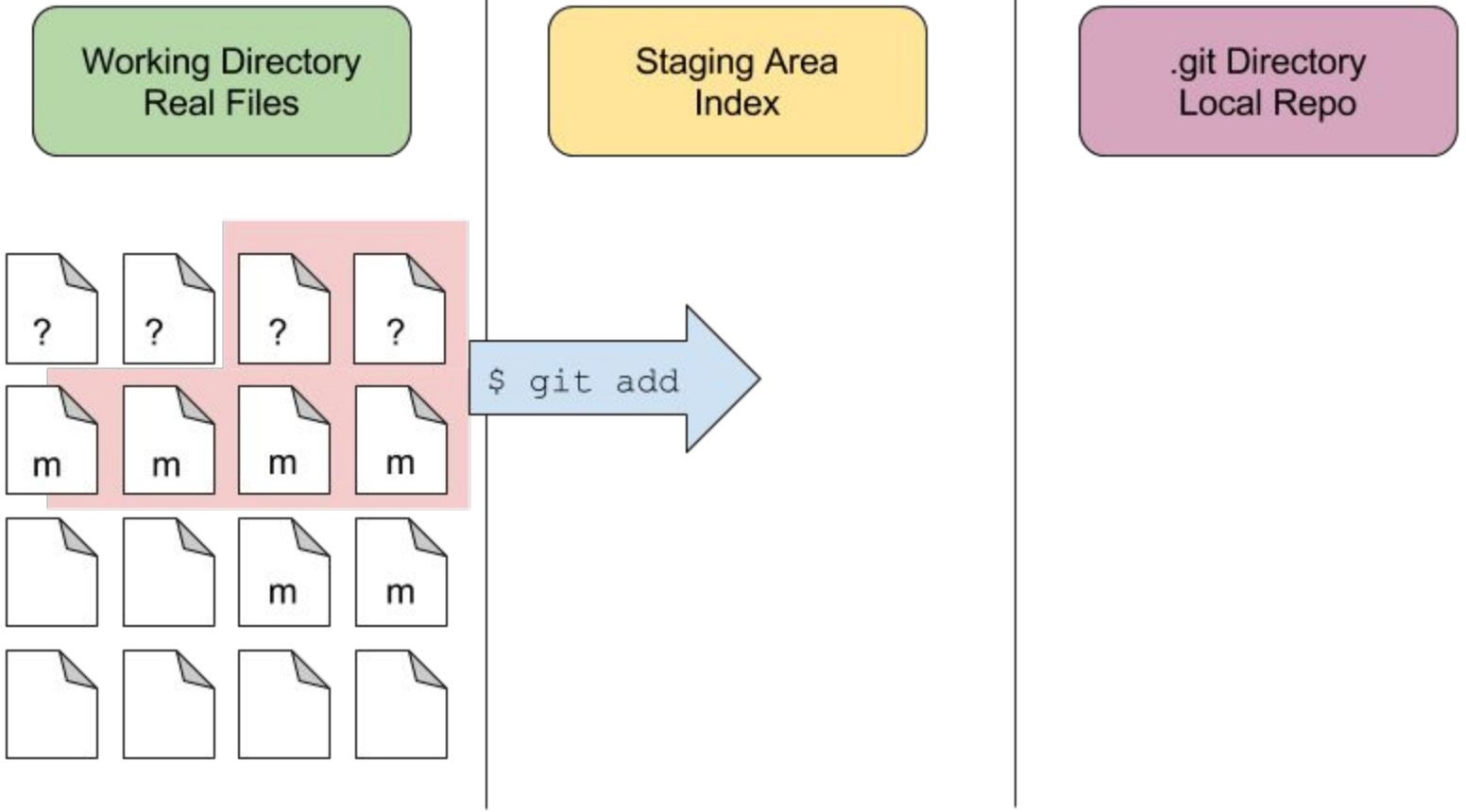
Working Directory
Real Files

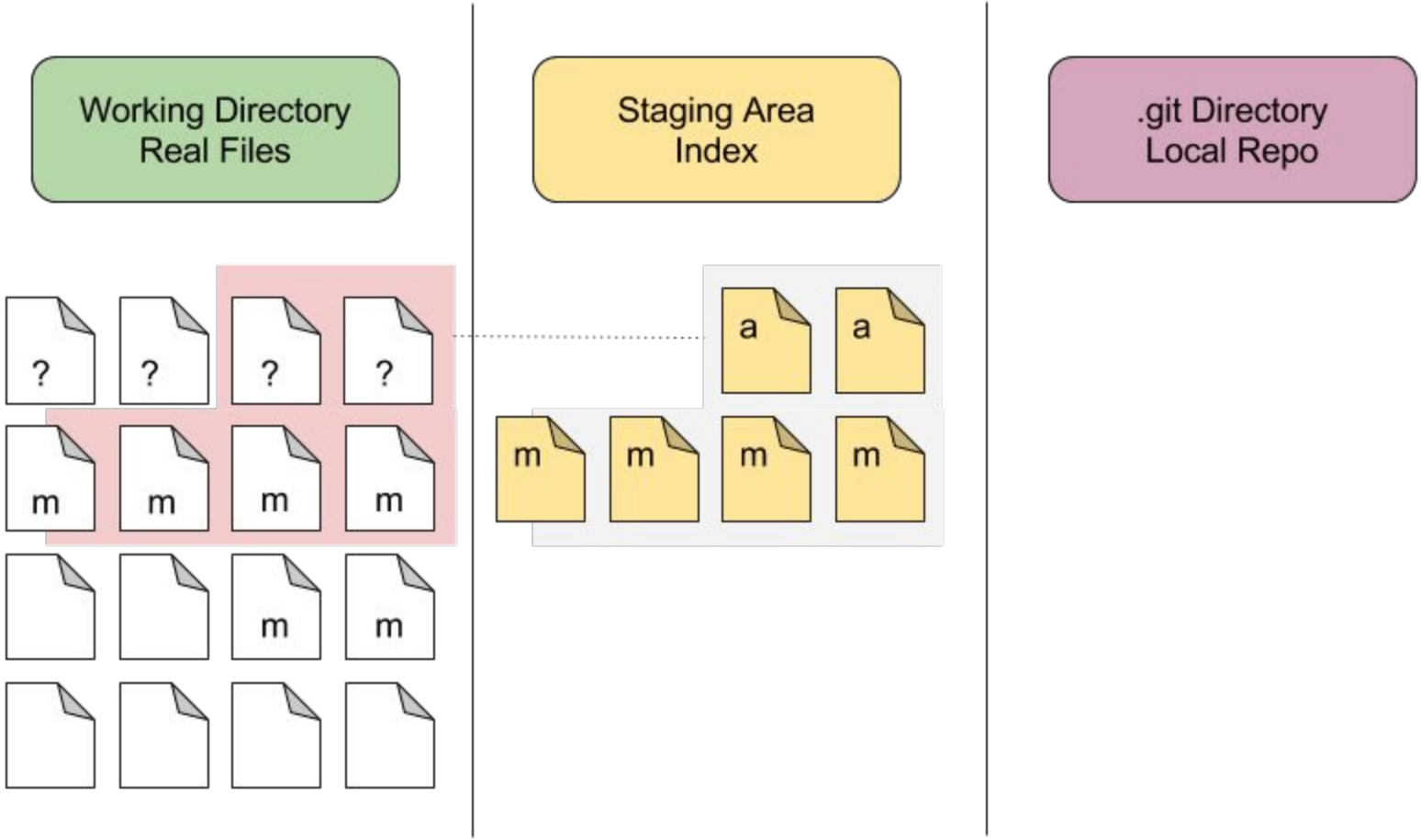
Staging Area
Index

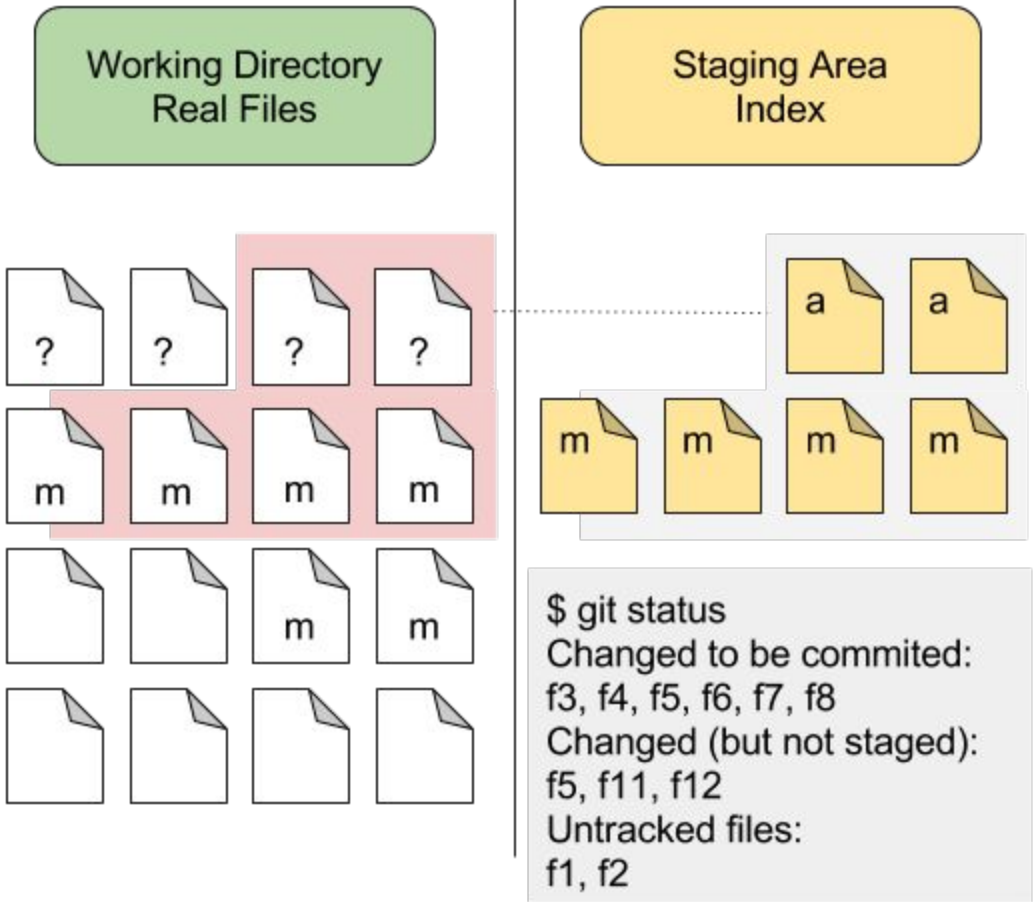
.git Directory
Local Repo

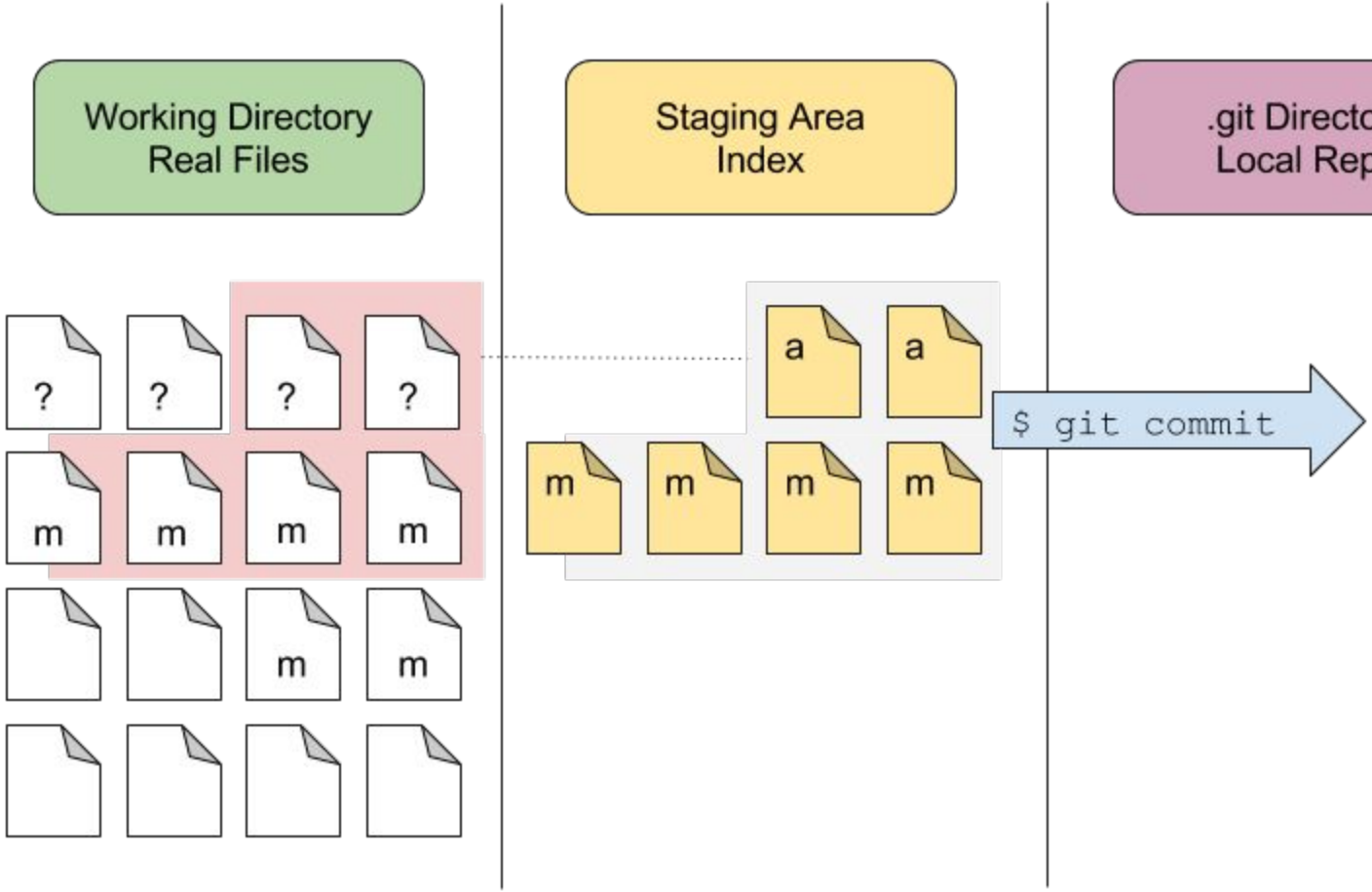



```
$ git status
Changed ( but not staged):
f5, f6, f7, f8, f11, f12
Untracked files:
f1, f2, f3, f4
```


Working Directory
Real Files
Staging Area
Index
.git Directory
Local Repo
? ? ? ?
m m m m
m m
$ git add


Working Directory
Real Files
Staging Area
Index
.git Directory
Local Repo
?
?
?
?
m
m
m
m
m
m
a
a
m
m
m
m

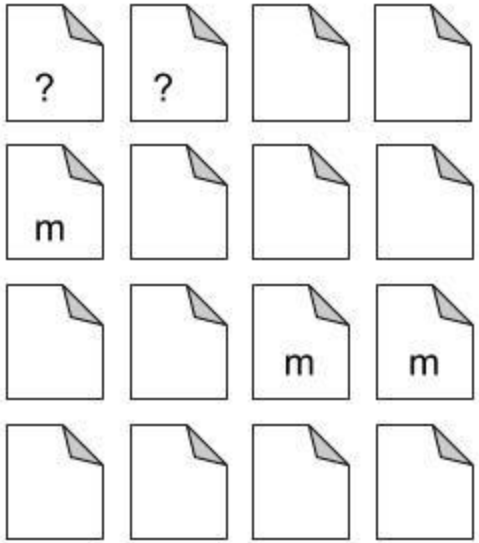

Working Directory
Real Files
Staging Area
Index
.git Directory
Local Repo
? ? ? ?
m m m m
m m
a a
m m m m
$ git status
Changed to be commited:
f3, f4, f5, f6, f7, f8
Changed (but not staged):
f5, f11, f12
Untracked files:
f1, f2


Working Directory
Real Files
Staging Area
Index
.git Directory
Local Repo
? ? ? ?
m m m m
m m
m m
a a
m m m m
$ git commit

Working Directory
Real Files

Staging Area
Index

.git Directory
Local Repo




```
$ git status
Untracked files:
f1, f2
Changed ( but not staged):
f5, f11, f12
```

Stage and Commit

Push file(s) to the staging area

```
git add <filename>
git add *
```

Commit the changes to the history in .git folder

```
git commit -m “Commit-Message”
```

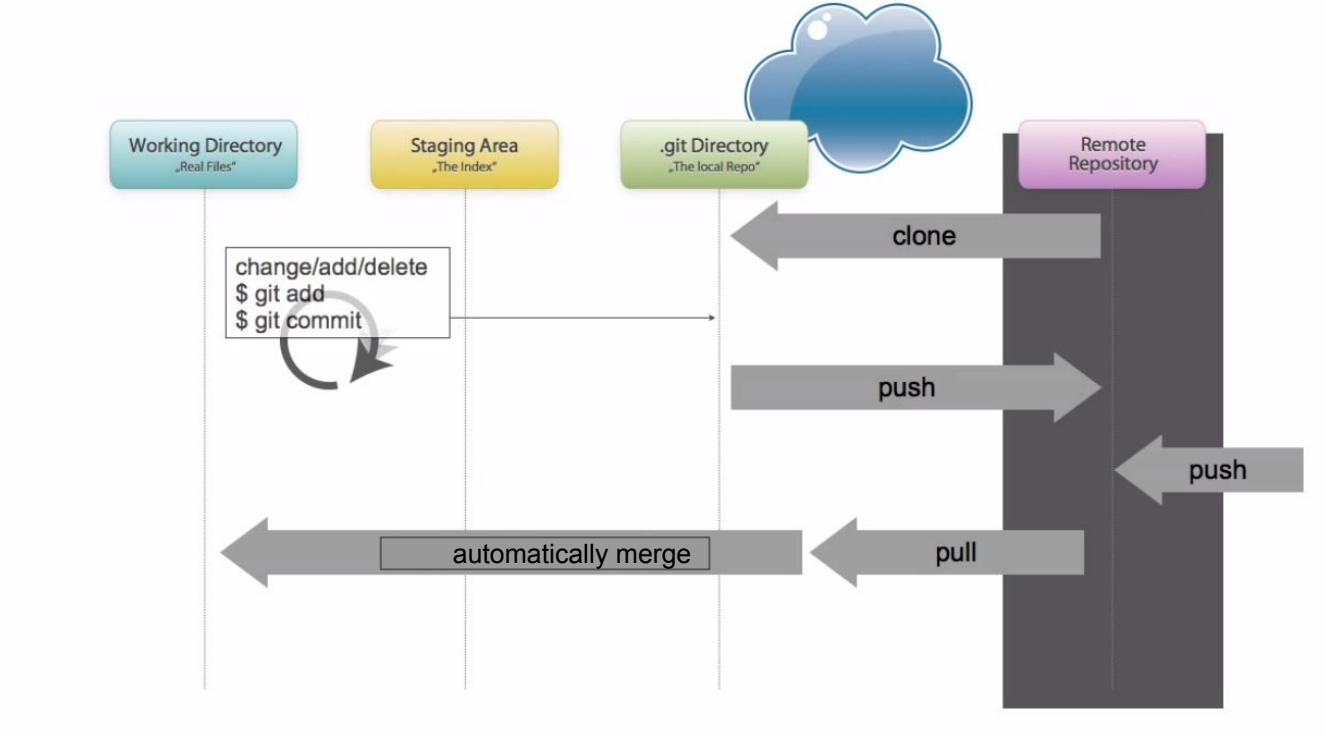
Log History

Show commit logs

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

git log --pretty=format:”%an: %h (%ar) - %S”
```

Remote -> Pull and Push



Live Demo

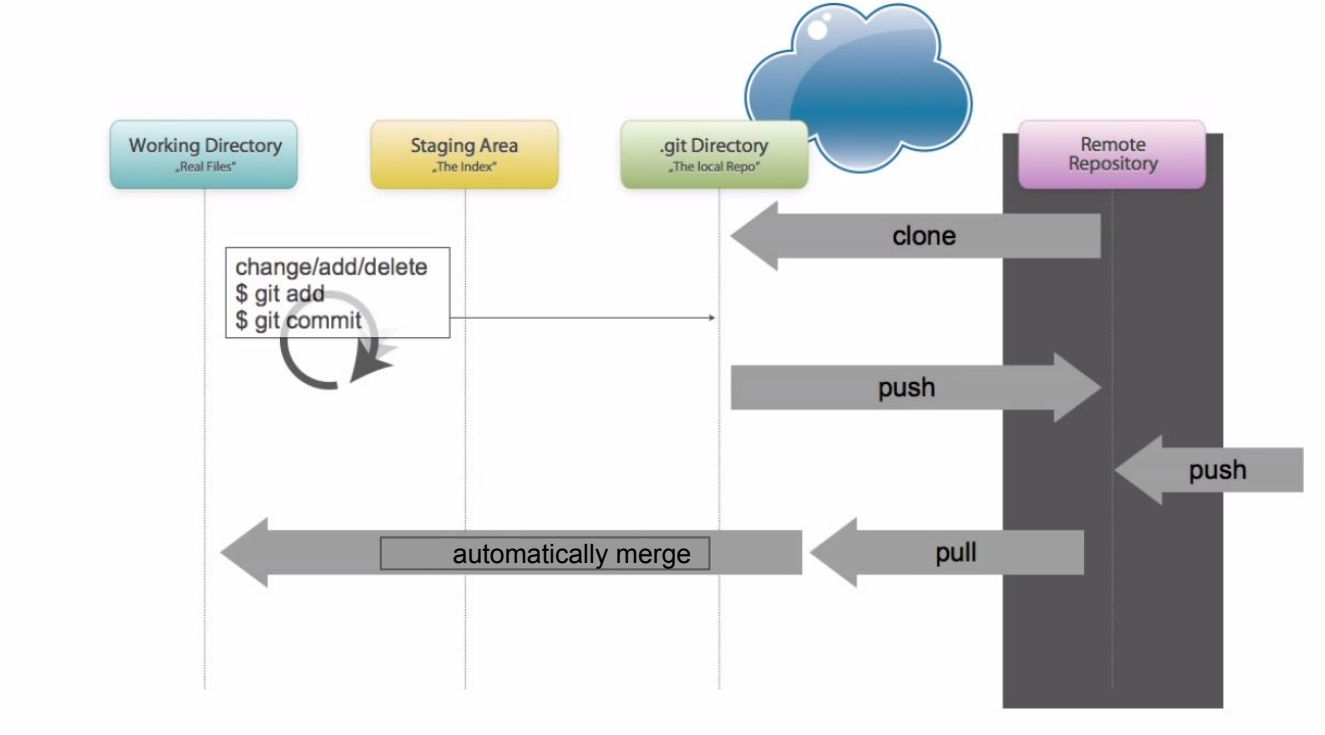
Default Settings GitLab

- Initialize your Tubit GitLab account
  - https://gitlab.tubit.tu-berlin.de/users/sign_in
  - Please enter your usual tubIT name and password and sign in
  - Let me know your name to invite you to the project
- Generating an SSH key
  - SSH keys are a way to identify trusted computers without involving passwords.
  - An SSH key allows you to establish a secure connection between your computer and GitLab.
    - cat ~/.ssh/id_rsa.pub
    - In case of no key
      - ssh-keygen -t rsa -C "email@example.com"
    - Copy-paste the key to the 'My SSH Keys' section under the 'SSH' tab in your user profile.

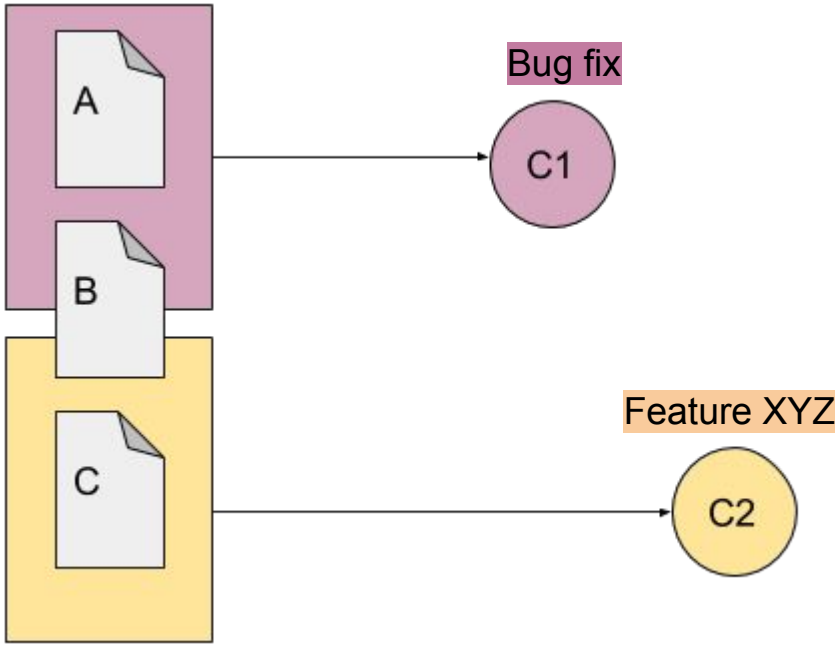
3. Session

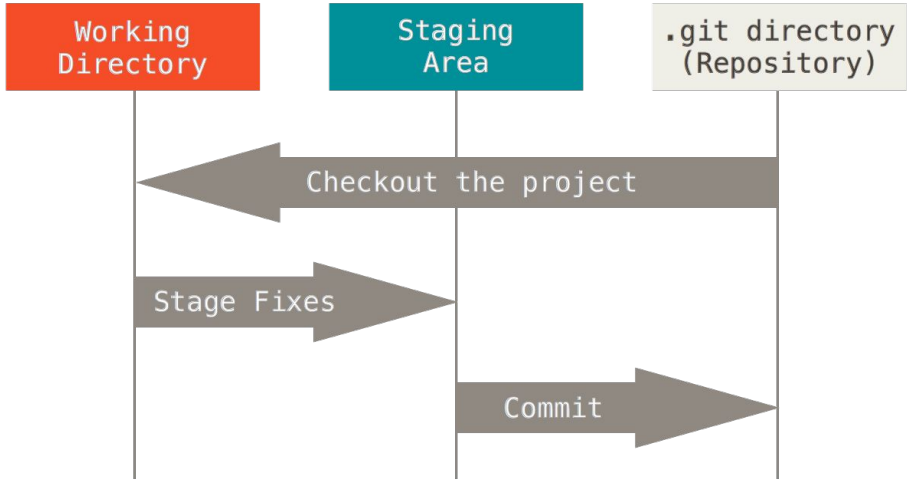
- Repetition
  - Local&Remote
  - Staging Granularity
- Exercise
- Branches
- Undo commits

Remote -> Pull and Push

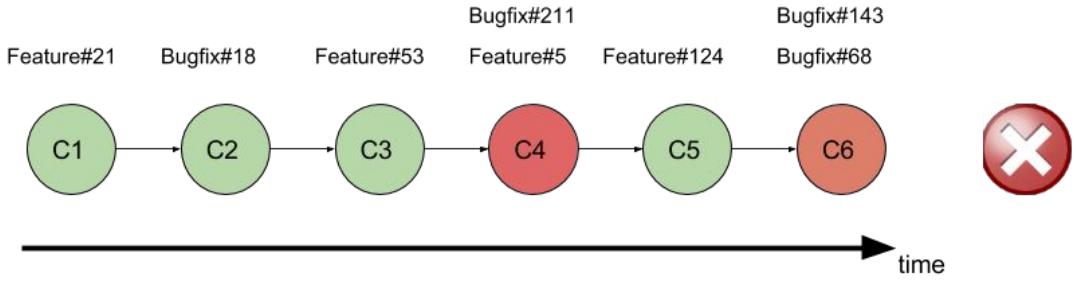


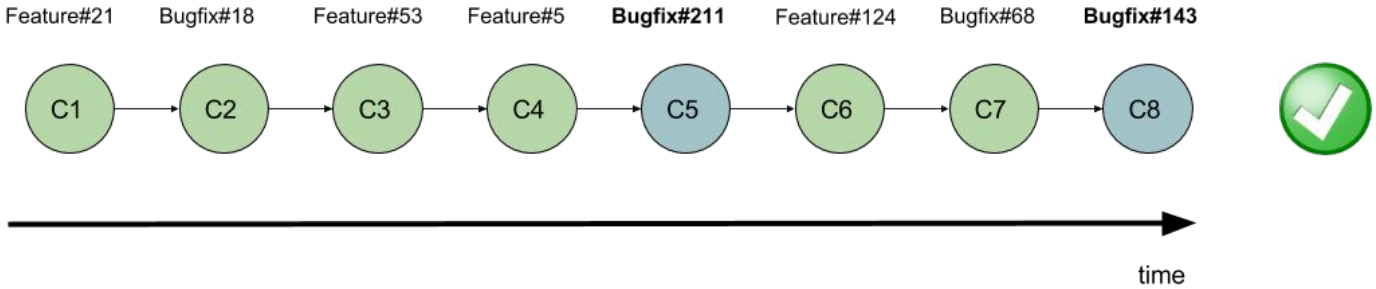
Advantage of Staging is Granularity





Granular commits





Task for Today

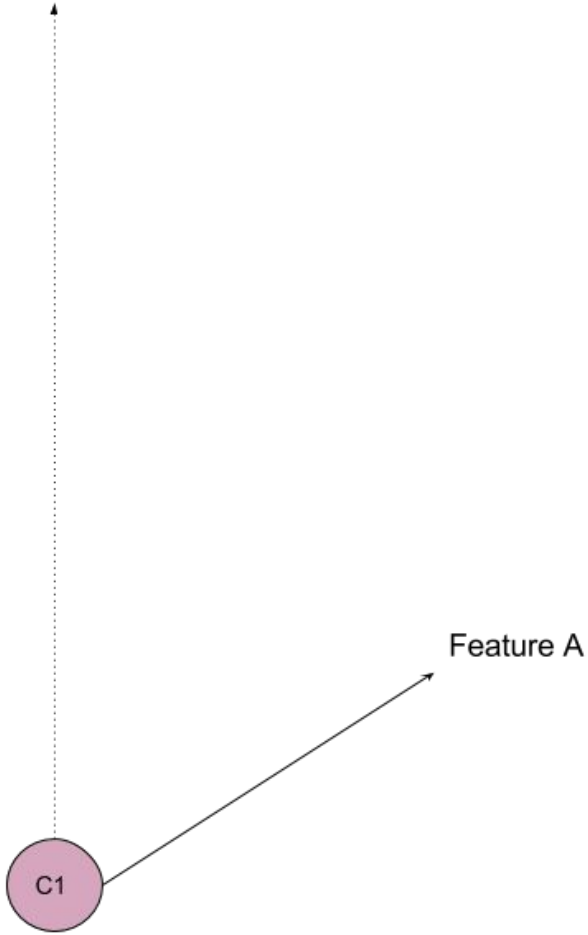
- Divide into teams (4x5 people)
- Develop a HTML website which gives you little information about a city
- You got an invitation from GitLab for the HTML project which has not finished up to now
- Each group takes care of only one city
- Complete the HTML Page

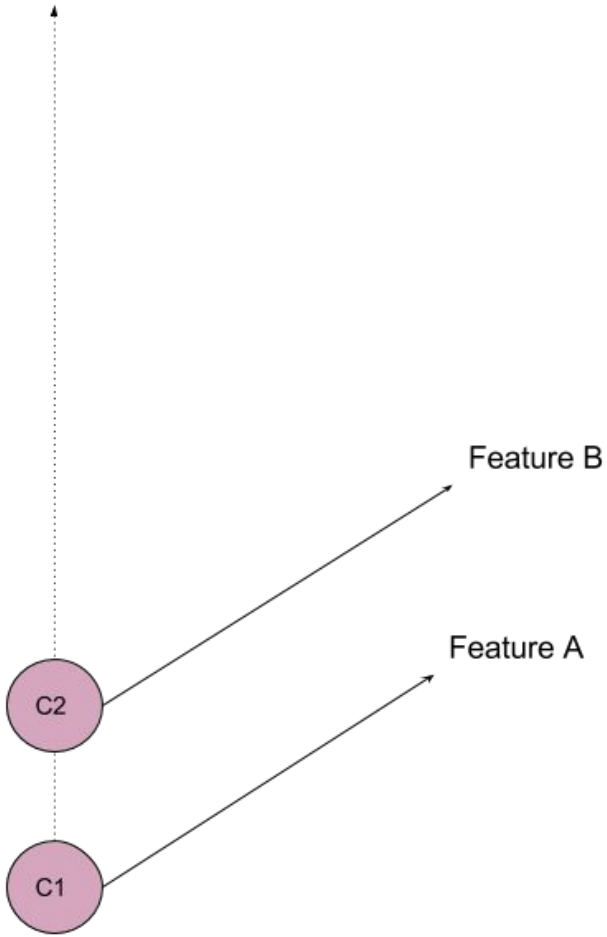
DELETE YOUR PROJECT ON YOUR LOCAL MACHINE AND CLONE THE PROJECT FROM SCRATCH

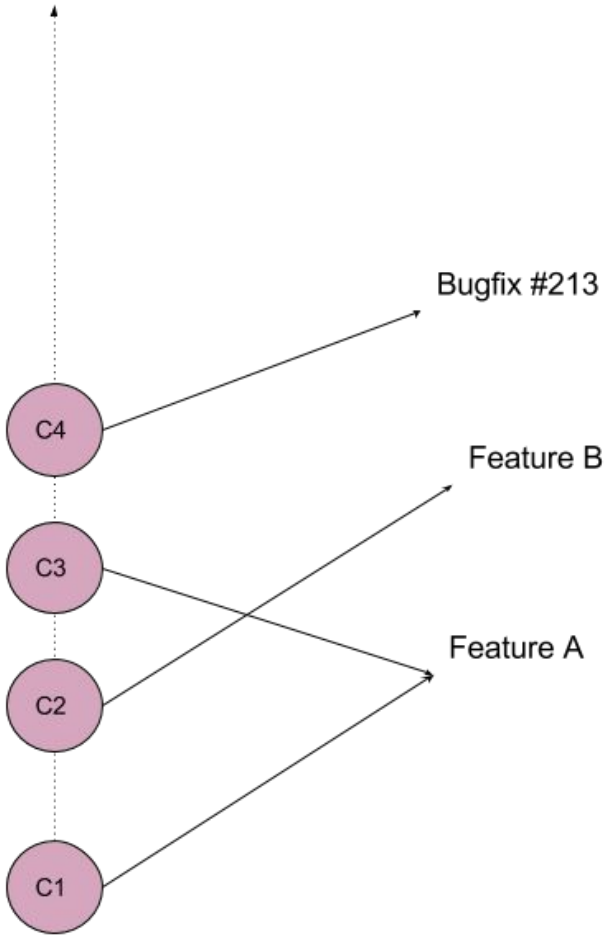
Steps

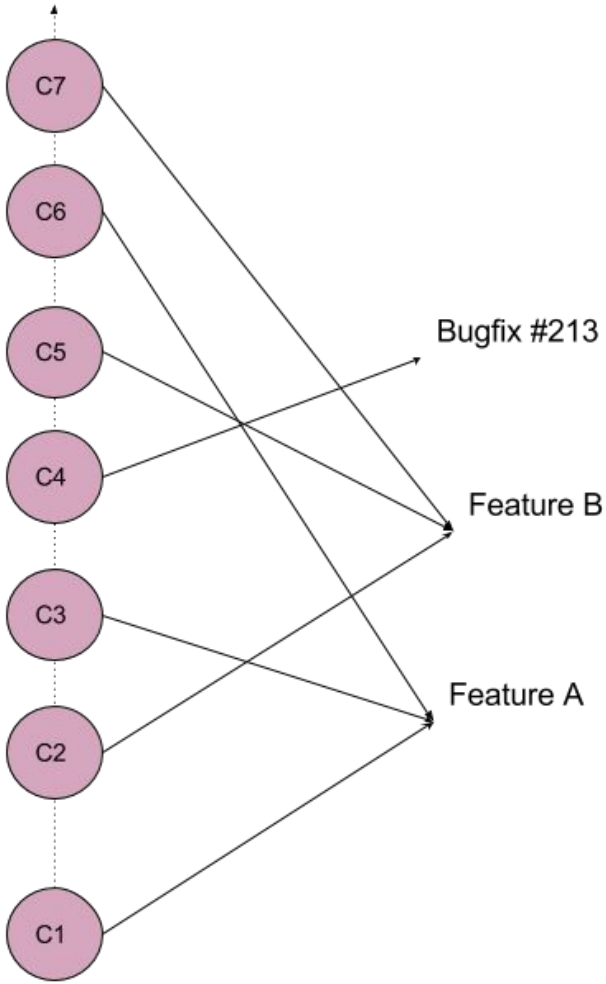
1. Clone the project from the remote repository
2. Find the project folder on your local computer
3. Copy the file template and rename it to your city
4. Change the values in your city HTML file
5. Copy the images into the folder ‘Images’
6. Link your file with the index.html
7. Check if the link works and the your city HTML looks fine
8. Pull the project from the remote to get the updated version
9. Add the HTML file to the staging area
10. Commit the staging area and add a meaningful comment to it
11. Add the image coat of arms to the staging area
12. Commit the coat of arms image with a meaningful comment
13. Add the Germany map to the staging area
14. Commit the staging area
15. Look into the history and see the commits
16. Push all your commits to the remote server
17. Look at GitLab if everything is correct

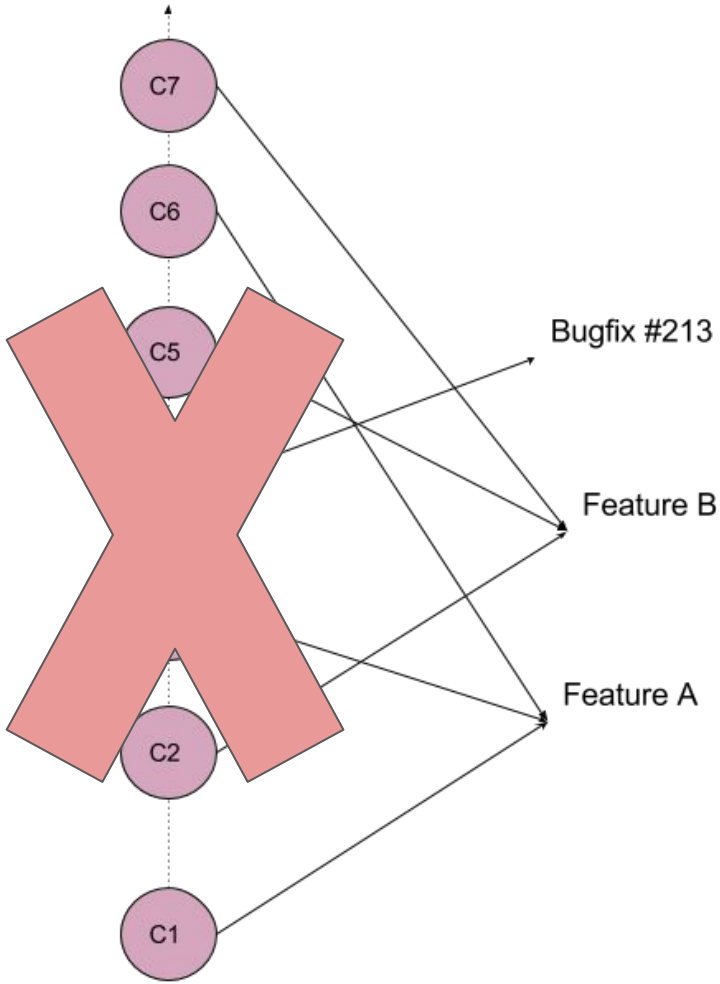
Branches

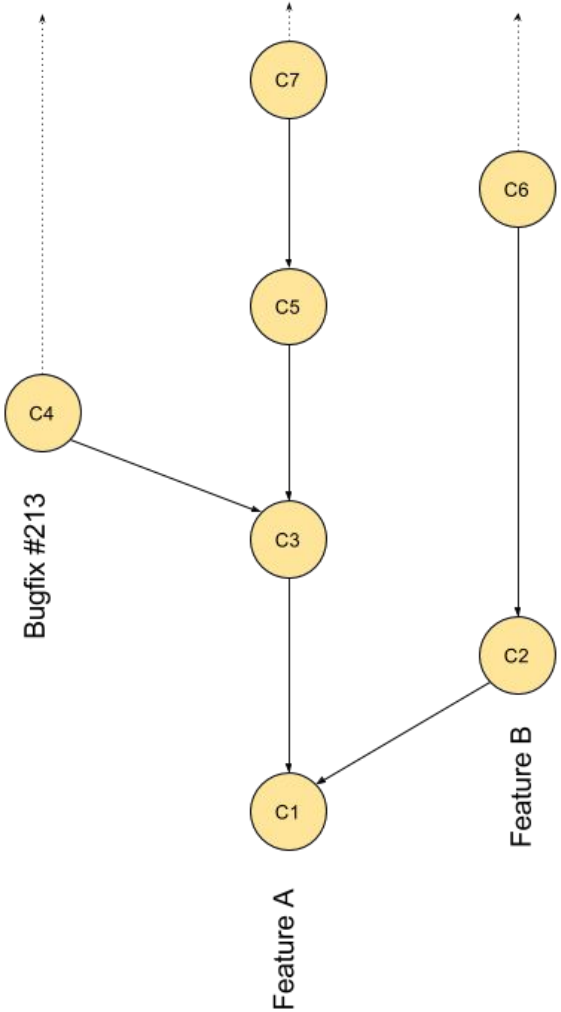

Feature A
C1

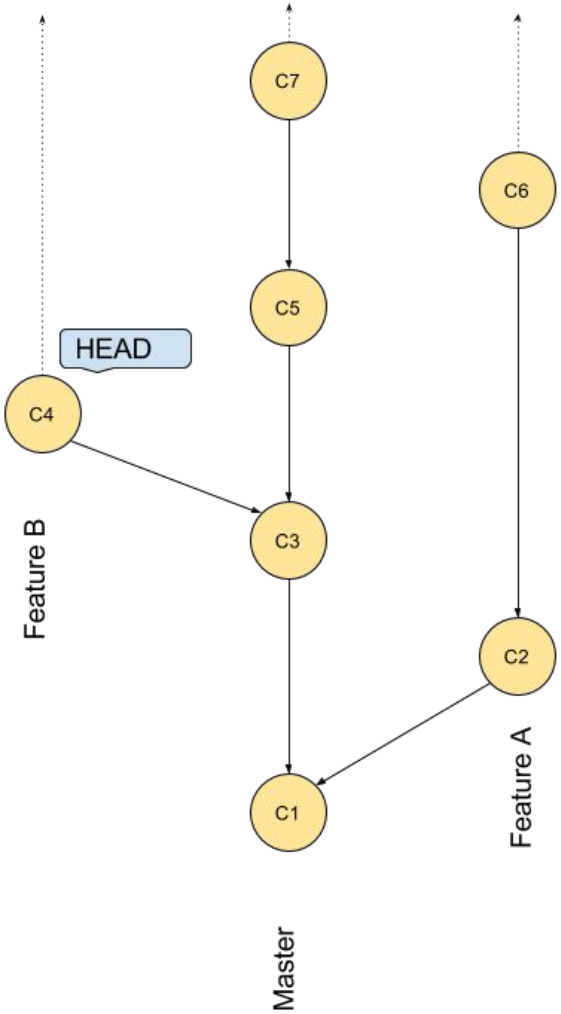

Feature B
Feature A
C2
C1

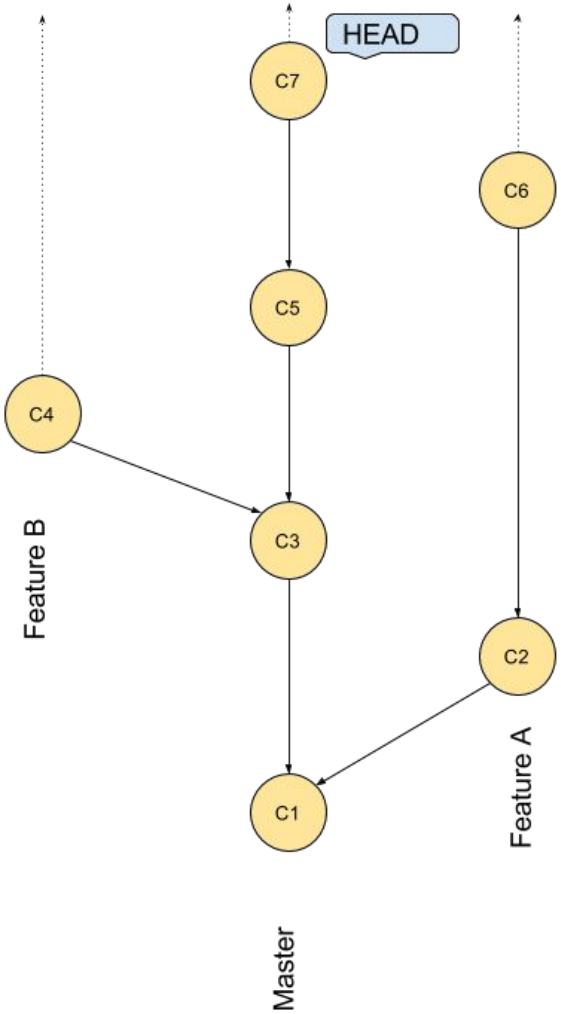

Bugfix #213
C4
Feature B
C3
Feature A
C2
C1


C7
C6
C5
C4
C3
C2
C1
Bugfix #213
Feature B
Feature A

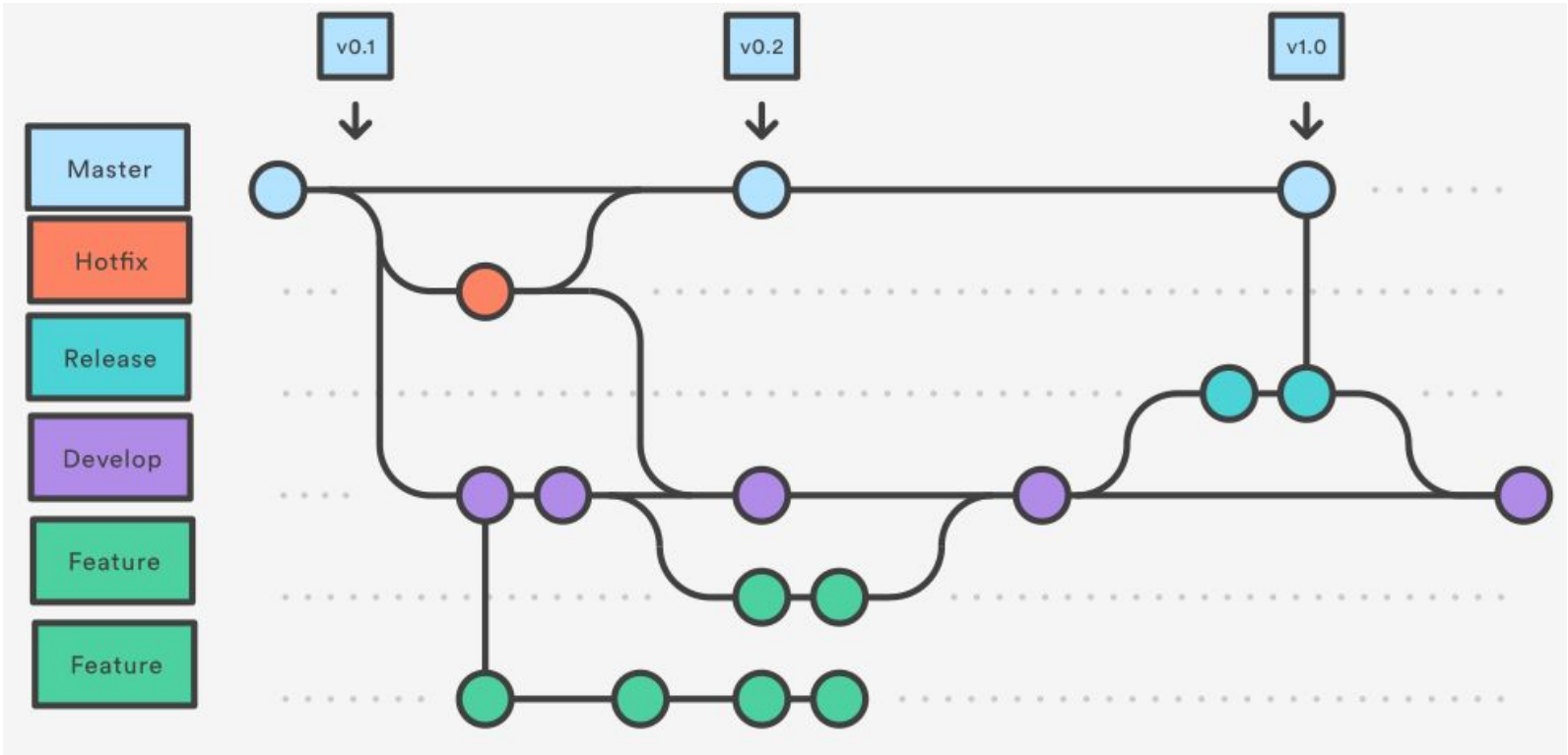

C7
C6
C5
C2
C1
Bugfix #213
Feature B
Feature A


C7
C6
C5
C4
C3
C2
C1
Bugfix #213
Feature A
Feature B


C7
C6
C5
HEAD
C4
C3
Feature B
C2
C1
Feature A
Master


HEAD
C7
C6
C5
C4
C3
C2
C1
Feature B
Feature A
Master

Git - Workflow



https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

Branches

It represents an independent line of development

It keeps the main master branch free from questionable code

Use branches for new feature or bugfix, no matter how big or how small

Merge the branch after completing with the master branch.

```
git checkout -b FeatureA
git checkout master
git merge FeatureA
git branch -d FeatureA
(git push origin FeatureA)
```

Live Demo

4. Session

- Undo commits
  - Revert
  - Reset
- Conflicts
  - Resolve Conflicts
- Git workflow for software projects
  - Feature Branch
  - Maintenance Branch
  - Release Branch
- Git conventions

Quiz

What is a commit?

What are the three steps to create a commit in Git?

What kind of repositories does exist in Git and what is the main difference between them?

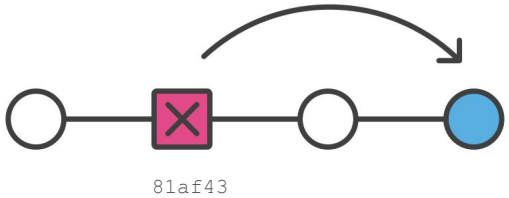
How you can identify a commit?

Undo Commits

- Revert method
- Reset method

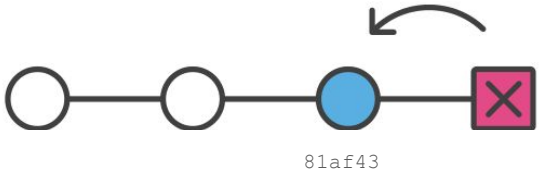
Undo Commits - Revert

- Revert
  - Instead of removing the commit from the project history, it figures out how to undo the changes introduced by the commit and appends a new commit with the resulting content.
  - Saves the integrity of your revision history
    - Prevents from losing history
  - `git revert 81af43`



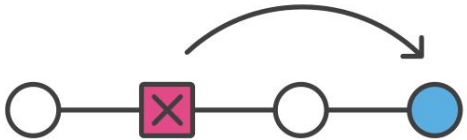
Undo Commits - Reset

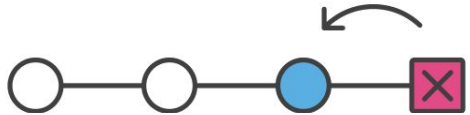
- Reset
  - It is a permanent undo
  - It should be used to undo local changes
    - never reset a commit that have been shared with other developers.
  - Versatile command
    - It removes commits
    - It undoes changes in the staging area and the working directory
  - `git reset 81af43`



Undo Commits - Revert & Reset

Revert “save method”	**Reset** “dangerous method”
Designed to safely undo a **public commit** because it doesn't change the project history.	**Designed** to undo **local commits**. Don’t use reset in the public history.
Undoes a **single commit**, it does not “revert” back to the previous state of a project by removing all subsequent commits.	Can only work backwards from the current commit -> **remove all** of the **commits** that occurred **after** the **target commit**
It is able to **target** an **individual commit** at an arbitrary **point in the history.** Tracking down a bug and find that it was introduced by a single commit.	**Remove** a committed **local commits** **Undo changes** in the **staging area** and **working directory**





Git - Conflicts

- Conflicts occur
  - if two or more people changed
    - the same lines in that same file
    - one person decided to delete the file while the other person decided to modify the file
  - Integrating another branch into your current working branch

Git can’t know what is correct.
He prints out a conflict message.

- You can always undo a merge and go back to the state before the conflict occurred.

Resolve Conflicts

- When you open the file in text editor, you will see both changes
- Git adds conflict markers to the affected areas
  - It begins with <<<<<< and ends with >>>>>>
  - Two conflicted blocks themselves are divided by a ======
- Options
  - Either you keep your changes
  - take your colleague changes
  - make a brand new change
- Commit this file after changing

```
1  <<<<<<< HEAD
2
3  Here is the original change.
4  =======
5  Here is the modified change.
6  >>>>>>> 58326c301d09b58f3ac23d616e73f7b478424cc5
```

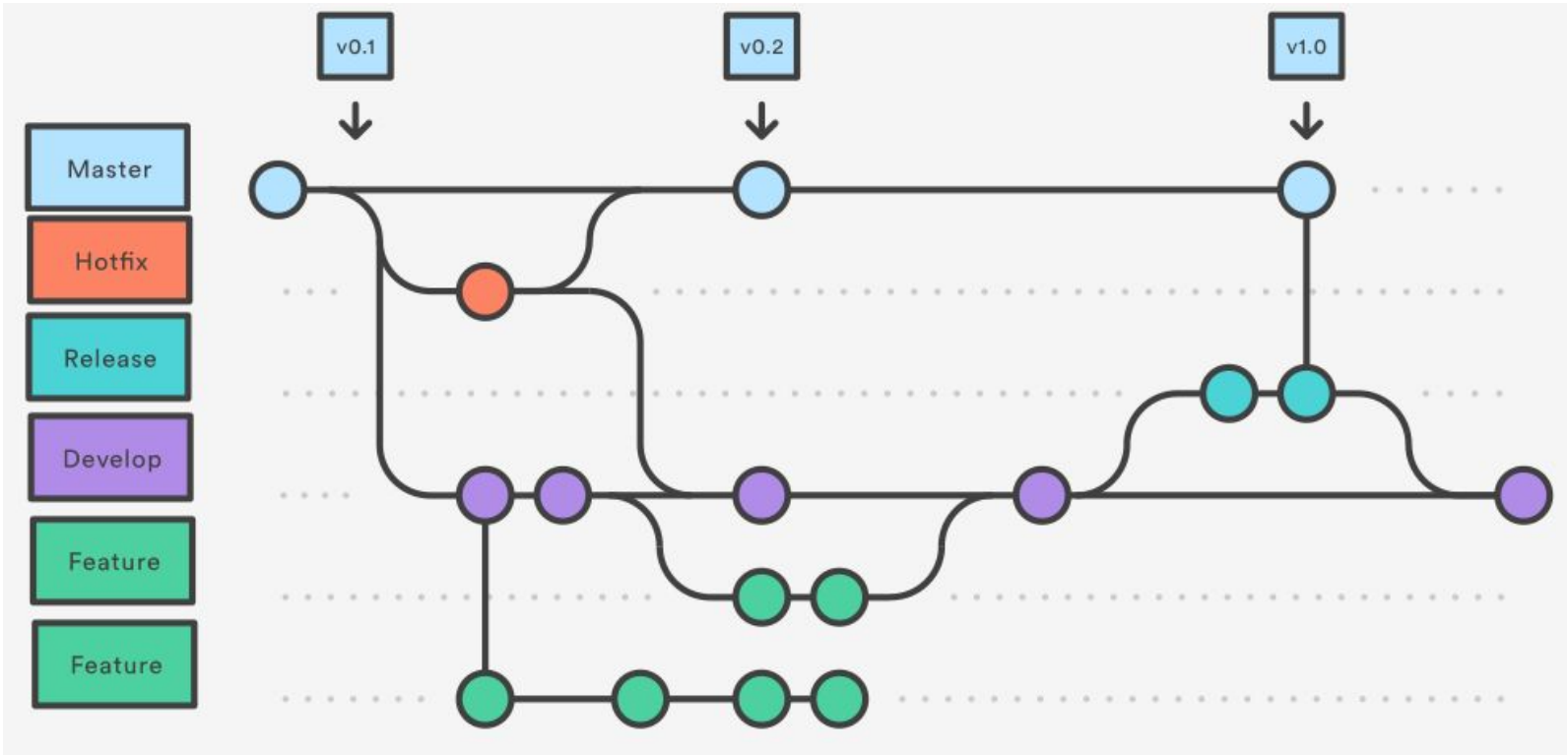

https://github-images.s3.amazonaws.com/enterprise/11.10.340/user/assets/images/mac/changes/merge_conflict_sample.png

Live Demo

Git Workflow for Software Solutions

Typically application for software projects

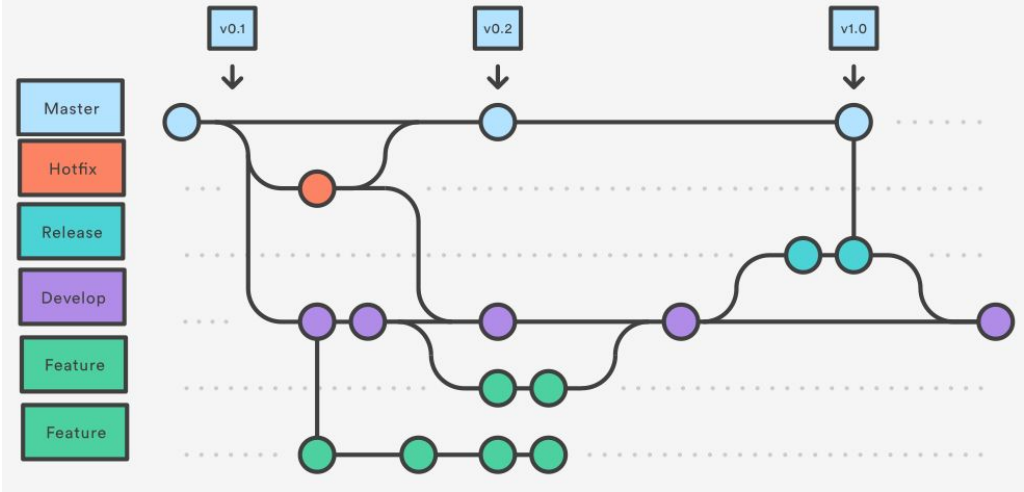
Git - Software Developing Workflow



https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

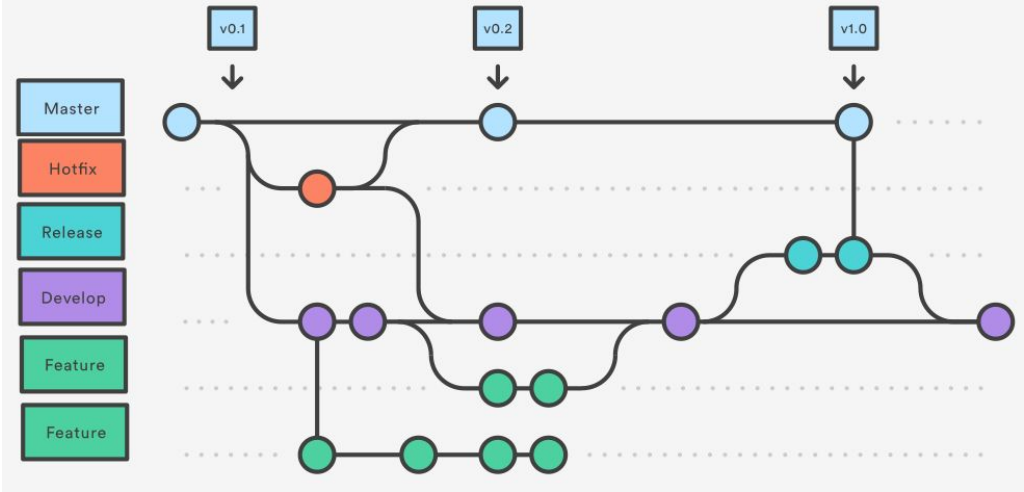
Feature Branch

- Each new feature **should reside** in its **own branch**
- Instead of branching off of master, feature branches **use develop** as parent **branch**
- When a **feature** is complete, it gets **merged** back **onto develop**
- **Features** should **never interact** directly with **master**
- **Combine** feature **commits** (squashes)
- **Delete branch** after **complete** feature



Maintenance Branch

- **Quickly** patch **releases**
- The **only branch** that can **fork directly** off of **master**
- After **fixing**, it should be **merged** into both **master** and **develop**
- **Master** should be **tagged** with an updated **version number**
- No **waiting** or **interrupting** for the next release cycle for crucial bugfixes

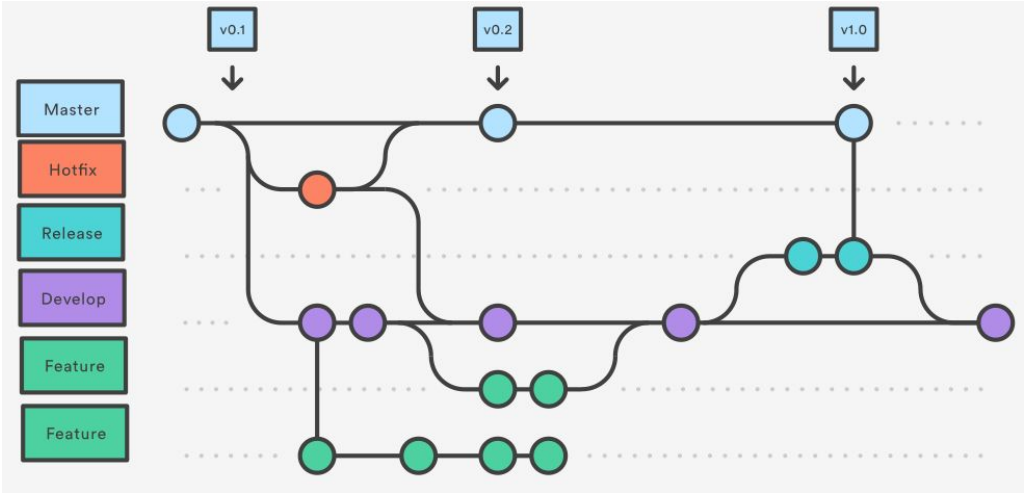


Release Branch

- Once
  - develop branch has **enough features**
  - a release **date** is **approaching**

  you fork a release branch off of develop.
- Creating this branch starts the next release cycle
  - **only bug fixes**
  - **documentation**
  - and other release-oriented task

  should go in this branch.
- The release gets **merged** into **master** and **tagged** with a **version number**.
- It should be **merged back** into **develop**, which may have progressed since the release was initiated.
- It makes possible for one team to **polish** the **current release** while another **team continues** working on **feature** for the **next releas**e.



Git Rules

- Always pull from repository before push to repository
- Use the concept of granularity and commit only related changes
  - Granularity Staging Area
    - 1 Bugfix = 1 Commit
      1 Feature = 1 Commit
- Use branches for separate feature development or bug fixing
- Be careful in deleting (reset) commits in the history
- Reset command for not published commits; revert published commits
- If conflicts appear, discus them with your team member
- Give commit comments a meaningful expression

Resources

https://www.atlassian.com/git/tutorials/

https://git-scm.com/docs/

Stash

Record the current state of the working directory and the index and gives you back a clean working directory.

```
git stash
git stash list
git stash pop
```