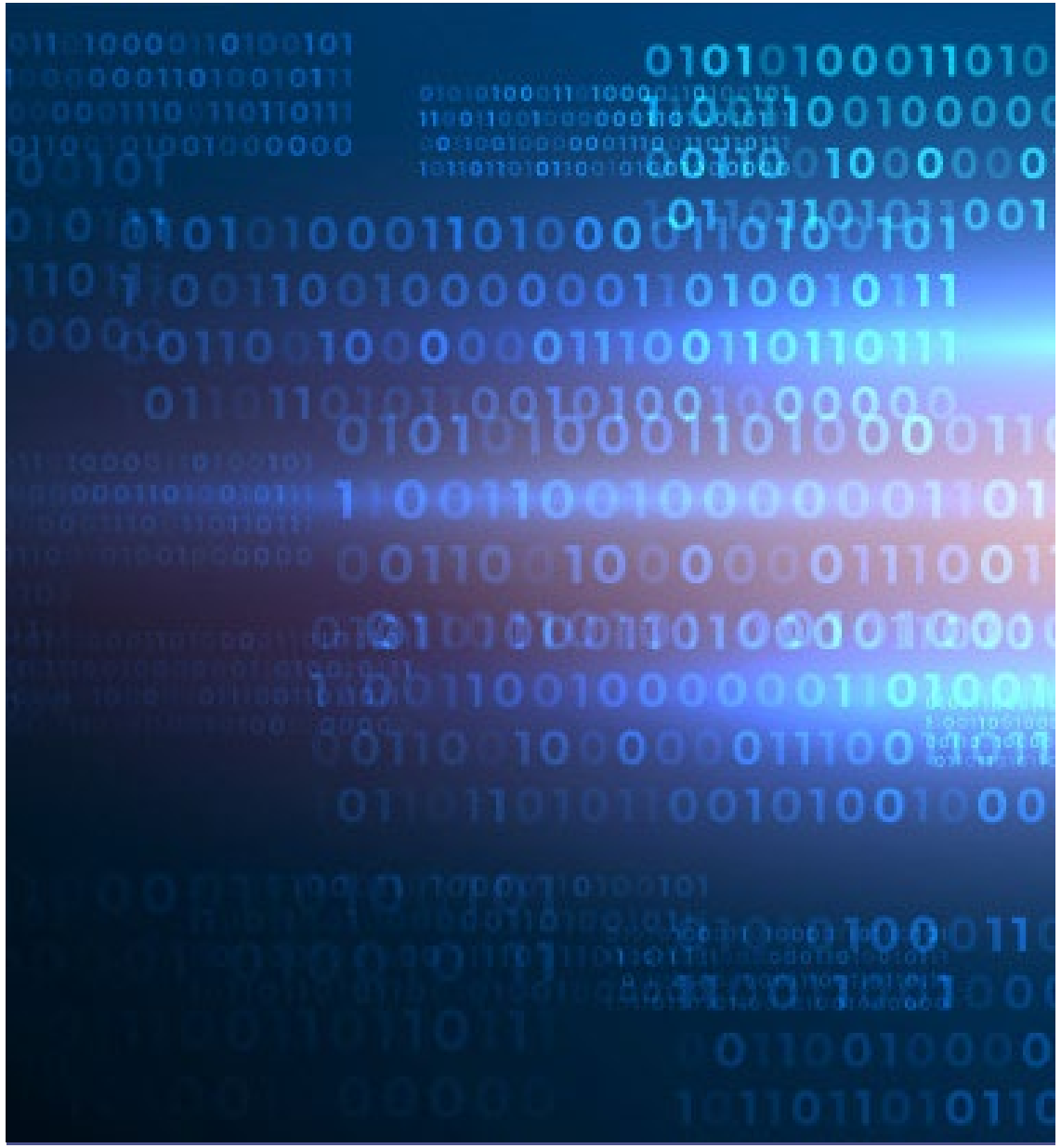
GIT

PRACTICAL VERSION CONTROL



AGENDA

- What is Git?
- Creating and cloning repositories
- Standard operations: Pull, Add, Commit, Push
- How git works
- Undoing commits, stashing and patching
- Resolving conflicts
- Branching, merging and rebasing
- Navigating and re-writing history

WHAT IS GIT?

1

WHAT IS GIT?

- Git is a Distributed Version Control system

It offers:

- Source code control
- Team working with a central repository
- Fast performance
- Easy branching and merging

WHY IS IT CALLED GIT? (WARNING – NSFW!)

- See https://git.wiki.kernel.org/index.php/Git_FAQ

Why the 'Git' name?

Quoting Linus: "I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'Git'".

('git' is British slang for "pig headed, think they are always correct, argumentative").

Alternatively, in Linus' own words as the inventor of Git: "git" can mean anything, depending on your mood:

- Random three-letter combination that is pronounceable, and not actually used by any common UNIX command. The fact that it is a mispronunciation of "get" may or may not be relevant.
- Stupid. Contemptible and despicable. Simple. Take your pick from the dictionary of slang.
- "Global information tracker": you're in a good mood, and it actually works for you. Angels sing and light suddenly fills the room.
- "Goddamn idiotic truckload of sh*t": when it breaks

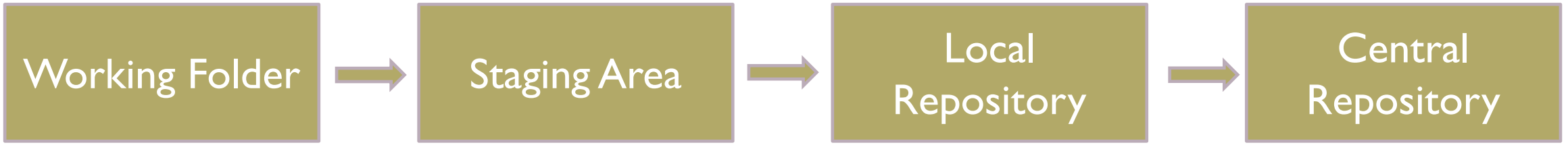
GIT PHILOSOPHY

- Git keeps complete versions of every file we store
- Every version of every file is given a unique hash value
- Every user has the full repository, and can make changes in isolation
- Repositories can be synchronized with a central repository

- Your company might have its own central repository, or use public ones like GitHub or BitBucket

GIT CONCEPTS

- The files that we can see and edit are in our working folder
- We can identify which new / deleted / changed files should be stored and these are added into a Staging area
- We then commit the changes to our local repository
- Optionally we synchronise with the central repository by pushing our changes



CREATING & CLONING REPOSITORIES

2

CLONING A REPOSITORY

- Cloning a repository makes a local copy of the repository on your machine
- The local copy includes all historic changes and commits, not just the current state at the point of taking the copy
- You will need to have a username and password to clone a private repository

```
git clone https://server/repositoryName

git clone https://username:password@server/repositoryName
```

ACTIVITY - CLONE A REPOSITORY

- Visit the website for the following GitHub repository https://github.com/vppmatt/paymentgateway
- Click on the Green CODE button and then click on the copy icon to select the git repostiory URL
- In a command prompt, navigate to a folder in your workspace and issue the command git clone followed by the URL you copied
- Review the files copied to your computer

- Note that you do not need a username / password to clone a public repository.

CREATING A REPOSITORY

- Use the git init command to create a local repository
- To create a remote repository:
  - create a new empty repository and then clone it, or
  - push a local repository *

* Follow the instructions on your repository server to guide you through it!

ACTIVITY - CREATE A REPOSITORY

- Create an account on GitHub if you don't already have one
- Create a repository – call it anything you like. It can be private or public.
- Select the option to add a readme file.
- When the repository is created, you can clone it locally.
- Optionally go to settings, then manage access to add collaborators – only collaborators can make changes to your project.

.GITIGNORE

- You can specify folders and files to exclude from the git repository using a .gitignore file

ACTIVITY - GITIGNORE

- Review the .gitignore file in the paymentsgateway project

WORKING WITH SECURE REPOSITORIES

- To avoid having to enter the username and password/token when pushing to a remote repository, include the username and password/token when cloning
- ```
  git clone https://username:password@server/repositoryName
  ```
- These details can also be saved in the .git/config file
  ```

STANDARD OPERATIONS

3

GIT COMMANDS

- As you make changes in your working folder, you will then want to send these into the repository. This is a 3 step process:

STEP 1 – add the changes to the staging area.

- The status command will show you what has changed
- The add command places any changes into a temporary holding area. Changes could include new files, deleted files or amended files. You can select specific files (or folders) or use . to indicate everything.
- You can run the add command multiple times before you get to the next stage

GIT COMMANDS

STEP 2 – commit your changes to the local repository

- The commit command will record all the changes
- The commit command places all the changes currently in the staging area into your local repository.
- The staging area is then clean for new changes.
- You should commit after every new feature / bug fix etc – a commit should reflect a single group of related changes.

GIT COMMANDS

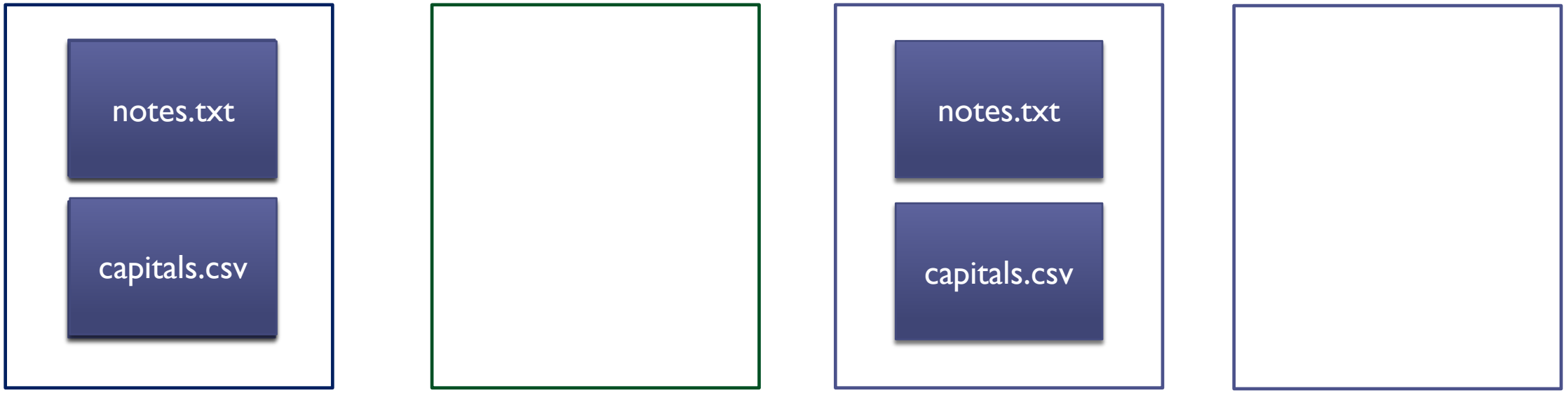
STEP 3 – push your changes to the central repository

- The push command will synchronise your changes with the central repository

BUT

- Before you run step 3 you must run the pull command to ensure any changes in the central repository are placed on your machine first.
- If there is a conflict when you run pull, that Git cannot solve, it will tell you which files need to be reviewed.
- If you make further changes, do an add + commit again.

GIT COMMANDS



Working area | Staging area | Repository | Remote Repository

```
git add . git commit -m "…" git push
```

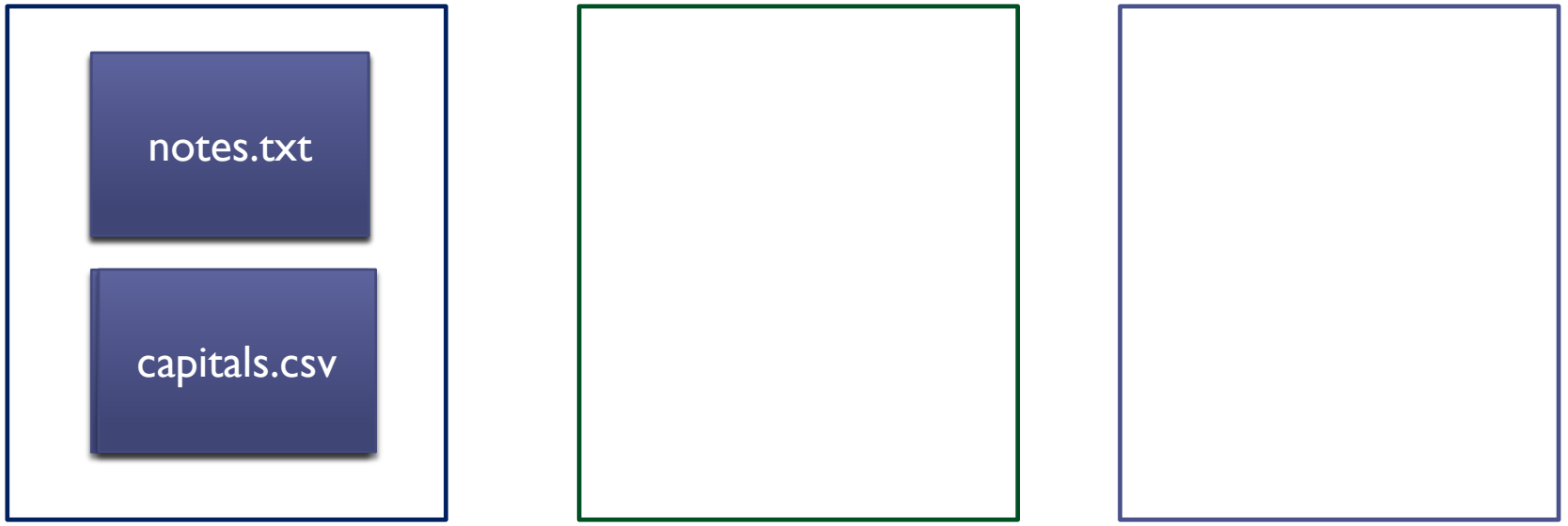
ACTIVITY - MAKING CHANGES

- Find out the hash value of the README.md file by running `git hash-object README.md`
- Make a change to the README.md file (add at least 1 new line and amend at least 1 existing line). Find out its new hash value (and check it's different)
- Create a new text file called about.txt and put your name in there.
- Run `git status` to see what files have changed
- Add the files to the staging area with `git add .`
- Commit the change with `git commit –m "some message"`
- Run `git status`
- Check for any remote changes with `git pull`
- Push your changes to the repository with `git push`

REMOVING AND RECOVERING FILES

- The rm command will remote a file from the staging area that has not yet been committed
- Use the --cached flag to keep a copy of the file in the working area
- Use the --f flag to remove the copy in the working area

GIT COMMANDS




```
git add .  git rm notes.txt
           git rm --cached capitals.csv
```

ACTIVITY - MAKING CHANGES

- Make a change to the about.txt file
- Run git status to see what files have changed
- Add the files to the staging area with git add .
- Remove the files from the staging area with git rm --cached about.txt
- Run git status
- Add the file to the staging area with git add .
- Run git status
- Remove the file from both the staging area and the working folder with Push your changes to the repository with git rm –f about.txt

HOW GIT WORKS

4

HOW GIT WORKS

- Each file and commit is given a hash value
- Git creates a compressed file for each hash value in the .git/objects folder
- When you commit changes, git does the following:
  - creates a compressed file for each added /amended file. Each file is named with its hash value
  - creates a compressed file (the "tree") listing all the files in the repository at the point of the commit, listing the hash values of each
  - creates a compressed file to record the commit, which lists the previous commit, the message, the author, date and time, and a reference to the tree file

HOW GIT WORKS

- You can view the contents of these files with git cat-file
- To view what happened in a commit, use git show
- To view the difference in a file between the current working folder and the staging area, use git diff
- To view the difference in a file between the staging area and the repository area, use git diff –cached
- To view the difference between 2 commits, use git diff <<commit1>> <<commit2>>
- It is normally sufficient to reference hash values with their first 4 characters

ACTIVITY - HOW GIT WORKS

- Find out the hash value of the latest commit using git log
- Find the location of the file relating to that commit
- View the contents of that file with git cat-file –p <<1st 4 digits of hash value>>
- Notice that the file references 2 other hash values. The parent is the previous commit. The tree is the list of changes. View each of these files
- When viewing the tree, you'll see a hash value for each object – view some of those too

ACTIVITY - HOW GIT WORKS

- Find out the hash value of the first commit you did using git log . You can use git log --oneline to list just the commit names
- View the impact of that commit with git show <<1st 4 digits of hash value>>
- Make a further change to the README.md file
- View the difference between the README file in the working folder and the version in the repository with git diff

ALL MIGHT NOT BE LOST - SCENARIO

Suppose you do the following:

1. Change a file, and add it to the staging area
2. Change the same file, overwriting the initial change, and add that to the staging area
3. Commit the change

Can you find out what the change was that you made in step 1, before step 2?

ALL MIGHT NOT BE LOST - ANSWER

Can you find out what the change was that you made in step 1, before step 2?

- Git will have created a compressed file for the change after step 1
- but it won't have recorded that hash file in the commit log structure.
- This means that you will have an "orphaned" hash file.
- There is no way within Git to find that file, but you can search for it

- Here's an example script to find orphaned files

  https://github.com/jameshfisher/git-orphaned-files

UNDOING COMMITS, STASHING AND PATCHING

5

UNDOING CHANGES

- You can remove changes that are in the staging area but not yet committed with git restore --staged <<filename>> or git reset
- You can remove changes that are in the working folder area but not yet in the staging area with git checkout <<filename>>

ACTIVITY - UNDOING COMMITS

- Make a change to the README file and add it to the staging area
- View the status of the repository
- Remove the change from the staging area with `git restore --staged`
- View the status of the repository
- Undo the change by recovering the file from the repository with `git checkout`

UNDOING COMMITS

- To revert the repository back to the position it was at a previous commit use git revert <<1st 4 characters of hash value>>
- This does not remove the commits from the log but puts the files back to the position they were in
- You will need to exit the command file with :q
- Delete commits (USE WITH EXTREME CAUTION) with git reset --hard <<1st 4 characters of hash value>> then push with the -f flag

ACTIVITY - UNDOING COMMITS

- Put the repository back to the position where it was when you first cloned it with git revert <<1st 4 characters of hash value>>
- View the status of the repository and the content of the README.md file

STASHING

Scenario:

- We have been making changes, which are staged, but we haven't finished
- We need to make DIFFERENT changes, starting from the current head of the repository
- We will set our current changes to one side by stashing them with git stash. This will put the working folder back to the position of the repository
- Later we can recover our changes with git stash pop
- And when we have finished with them, git stash drop

ACTIVITY - STASHING

- Add a new file called about.txt, and make a change to the README file and add these to the working folder.
- Temporarily save these changes and revert back to the repository head with git stash
- Add a new file called hello.txt, and commit the file
- Now restore the changes from the stash with git stash pop
- Commit the changes
- Check the stash with git stash list, and then remove it with git stash drop

PATCHING

Scenario:

- You have changed a file but you want to add the changes as 2 separate commits as they relate to different features
- Use the --patch flag when adding the file to the staging area
- When prompted for each change, use ? to get an explanation of the options

ACTIVITY - PATCHING

- Make changes in 2 different places to the README.md file
- Add the file to the staging area using the --patched flag
- Stage and commit the 2nd change only
- Now commit the first change

RESOLVING CONFLICTS

6

RESOLVING CONFLICTS

- A conflict means that changes you have made overlap with changes made elsewhere
- A conflict can occur if your working folder is not up to date with the repository
- Typically conflicts occur in the following scenarios:
  - You have recovered files that were stashed, and the changes being made affect changes also made in the last commit(s)
  - Someone else has changed a file and pushed it to the remote repository. You are now changing the same file. In this instance, you can't do the push, you need to do a pull first.
- A conflict only occurs if you have altered the same lines in the file

UNDERSTANDING THE MERGE

```
<<<<<<< Updated upstream
Instructions Yesterday was Friday
=======
Instructions today is Saturday
>>>>>>> Stashed changes
```

Version currently committed in the repository

Version you have changed but hasn't been commited

- Git will not let you commit a merged file which has been unchanged

ACTIVITY - CONFLICTS

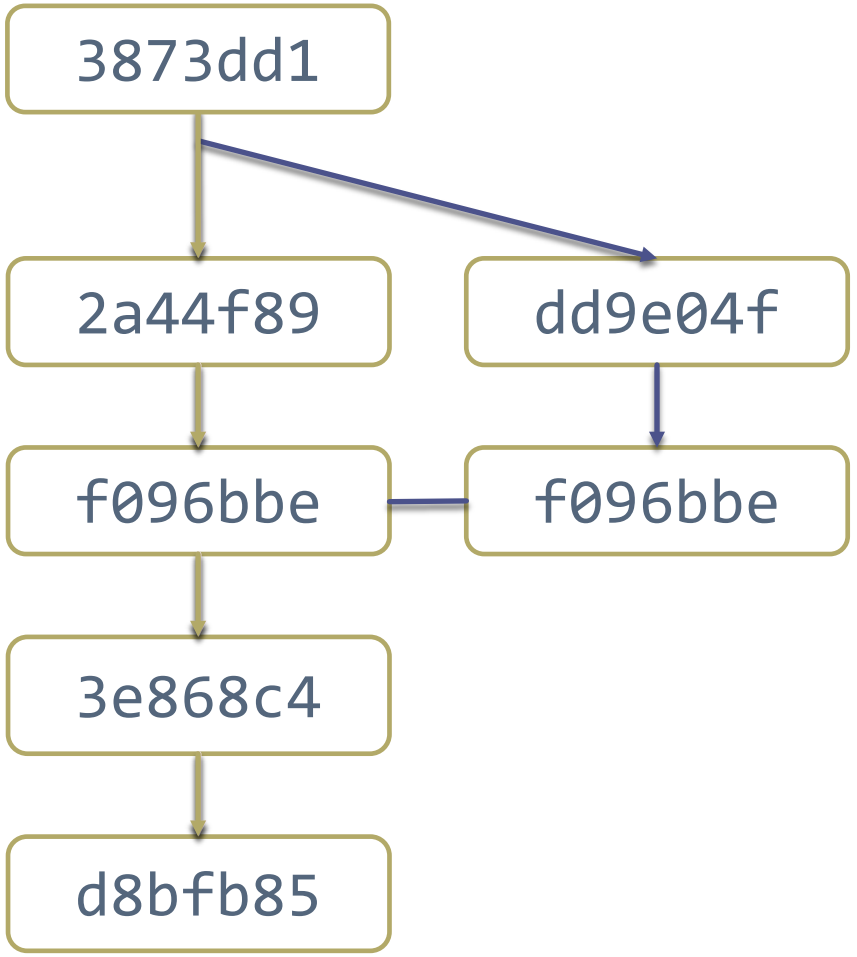
- Make a change to the README file and add it to the working folder.
- Temporarily save the change and revert back to the repository head with git stash
- Make a different change to the same part of the readme file and commit the change.
- Now restore the changes from the stash with git stash pop
- You will be notified of the conflict. Edit the README file and decide which version to keep before committing the change

BRANCHING, MERGING AND REBASING

7

BRANCHING

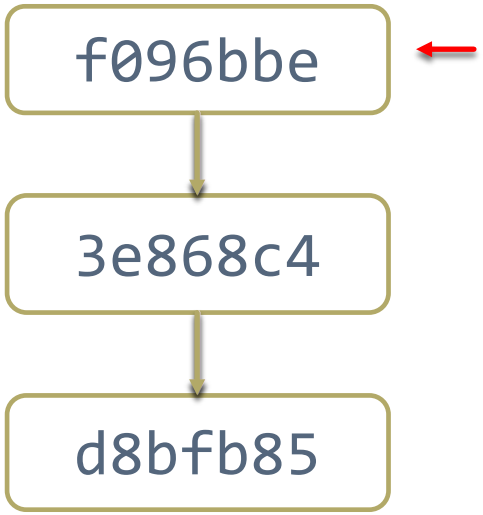
master branch



"a duplicate of one or more objects that allows development to take place on these duplicate copies leaving the original versions intact"

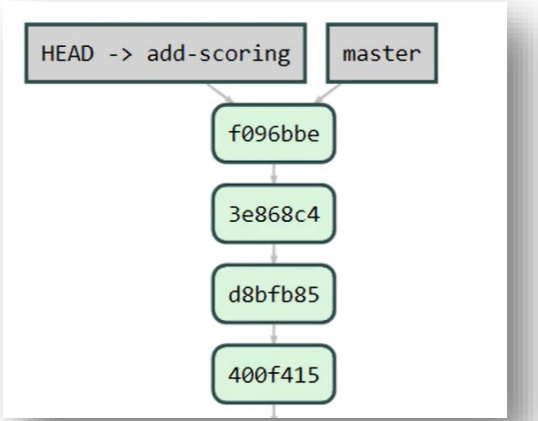
DETATCHED HEADS

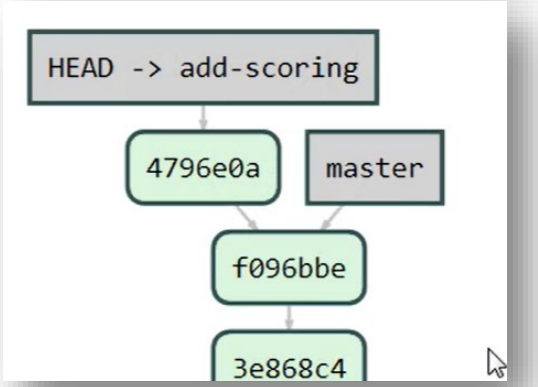
- The HEAD is a pointer to a commit. By default the HEAD is the latest commit.
- Our workspace will contain the files at the commit where our HEAD is
- When the HEAD is not at the latest commit, this is known as a detached HEAD state.
- The command git checkout HEAD~1 will step back 1 place in the branch
- The command git checkout master will move the HEAD to the last commit of the master branch (the main branch)



FEATURE BRANCHES

- Typically we create branches to develop separate features for our applications
- We create branches with git checkout –b <<branch-name>>
- The head will point to the new branch
- Git stores info about branches in the refs folder
- Changes are committed only to the currently active branch
- The git show-branch command will give a visual layout of the branches



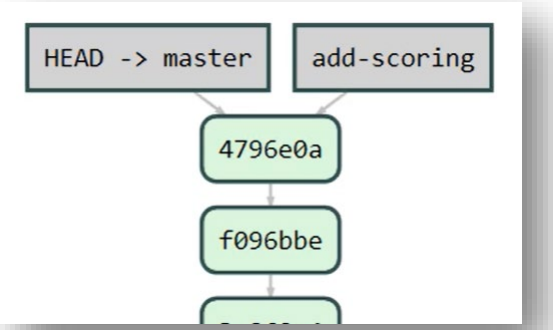


ACTIVITY - BRANCHES

- Create a new branch called html-redesign
- Make a change to the HTML in the html file in the root folder
- Commit the change
- Use git checkout master to switch to the master branch and git checkout html-redesign to switch back to the new branch

MERGING

- When our changes are ready to be finished, we can apply them to the main branch by merging them.
- First we switch to the main branch
- The git branch --no-merged command lists all branches that have not been merged back into the master branch
- To merge a branch back into the master, use git merge <<branch-name>>
- To remove a branch use git branch –d <<branch-name>>



MERGING SCENARIOS

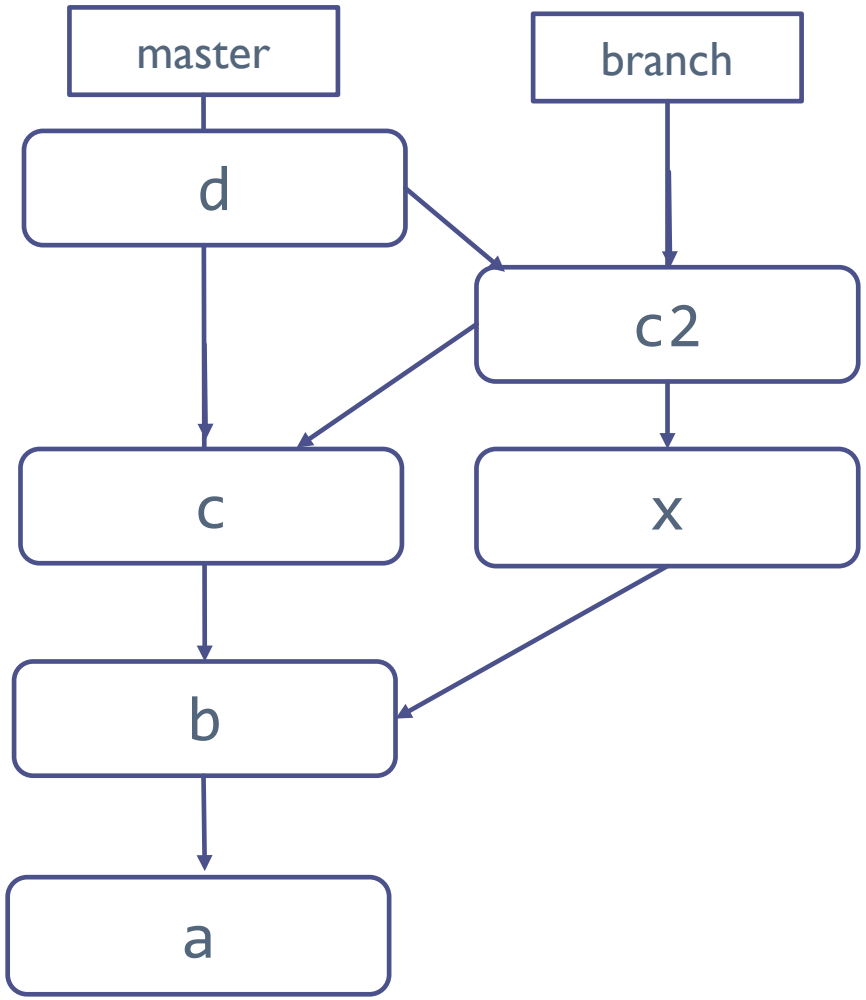
- A fast forward merge takes place when all that is needed for the merge is to move the position of the final commit of the master up a tree
- A recursive strategy merge will use a common ancestor and generate an additional commit in the master branch
- Where there are conflicts, Git will warn you and mark the conflicting lines for you to fix

ACTIVITY - MERGING

- Merge the html-redesign branch back into the master branch
- Within the html-redisgn branch, make a further change to the HTML in the html file in the root folder and commit the change
- Within the main branch, make a change to the README.md file and commit the change. Also make a change to the html file – amend the same line as before.
- Merge the html-redesign branch into the master branch.
- Use git diff to see the conflicts
- Resolve the conflicts and commit to finish the merge

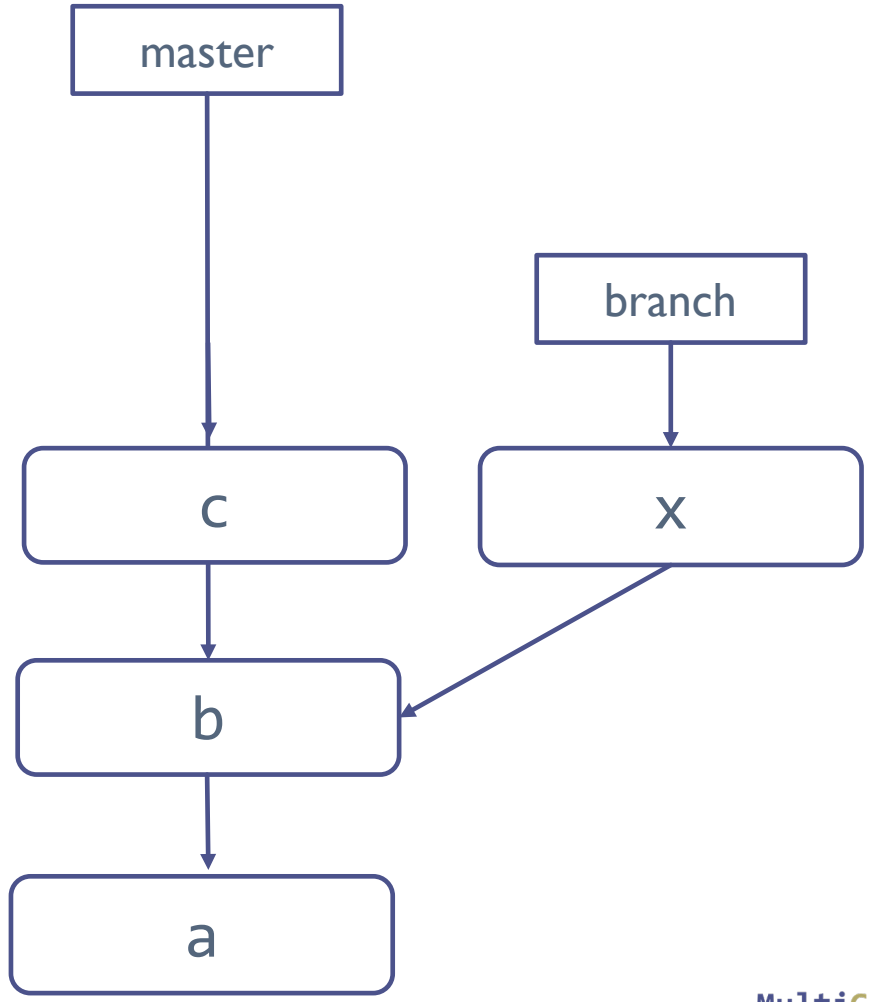
REBASING

- Rebasing allows us to change the order of commits
- It provides us with an alternative way to allow us to apply commits in the master branch to our feature branch
- Without rebasing, our log becomes messy and difficult to trace for errors



REBASING

- Rebasing changes the start position of a feature branch
- The command is git rebase <<commit>> or git rebase master to rebase to the final commit in the master branch
- This allows the final merge from the branch back into master to be a fast-forward merge
- It also ends up with a cleaner graph!



ACTIVITY - REBASING

- Make a new branch called further-redesigns
- Within the new branch make a change to the HTML file and commit it.
- Within the master branch, make a change to the HTML file and commit it – amend the same line as before.
- Rebase the branch to commence with the latest commit in the master branch with git rebase master
- Resolve any conflicts and commit if needed to finish the rebase
- Make a further change in the branch, then merge back into master

NAVIGATING AND REWRITING HISTORY

8

NAVIGATING

- To move the working folder to a previous commit FOR VIEWING we can use git checkout <<commit>>
- You can also use a branch name with a symbol eg git checkout main~2
- To move the working folder and the HEAD back to a previous commit TO UNDO COMMITS use git reset --hard <<commit>> on a branch which has not been pushed, or git revert HEAD
- Symbols which can be used after a branch name:

^ or ~1	previous
~2, ~3 etc	back 2 or 3 commits…

REWRITING HISTORY

- Only change history which you haven't yet shared
- You can change the message in a commit with git commit --amend -m "new message"
- Changing a commit message will create a new commit with a different hash and remove the old one from the branch
- We have seen that you can undo a commit with git revert HEAD or git reset –hard. It is safe to use git revert HEAD– this is adding a new commit (but git reset --hard isn't – so only use it if you haven't pushed changes!)