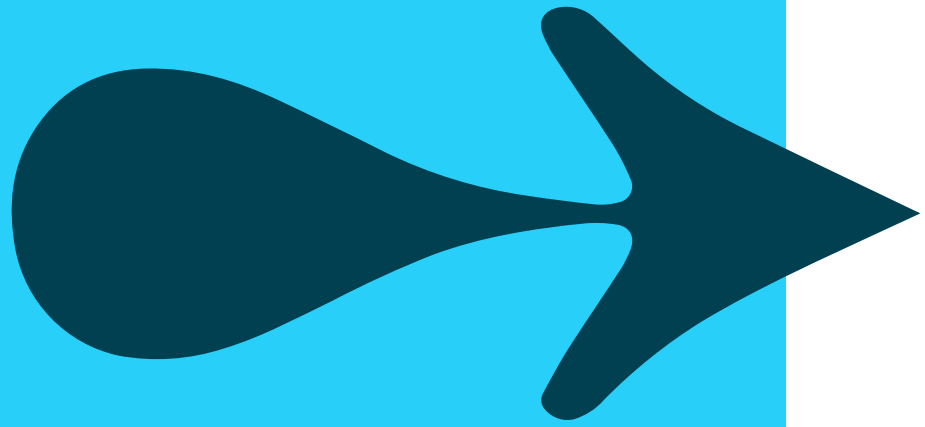
POWERING POTENTIAL

Version Control Using GIT

GIT BASICS

Version control

- CVCS
- DVCS
- GIT



Version Control

Version control is a system that records changes to files

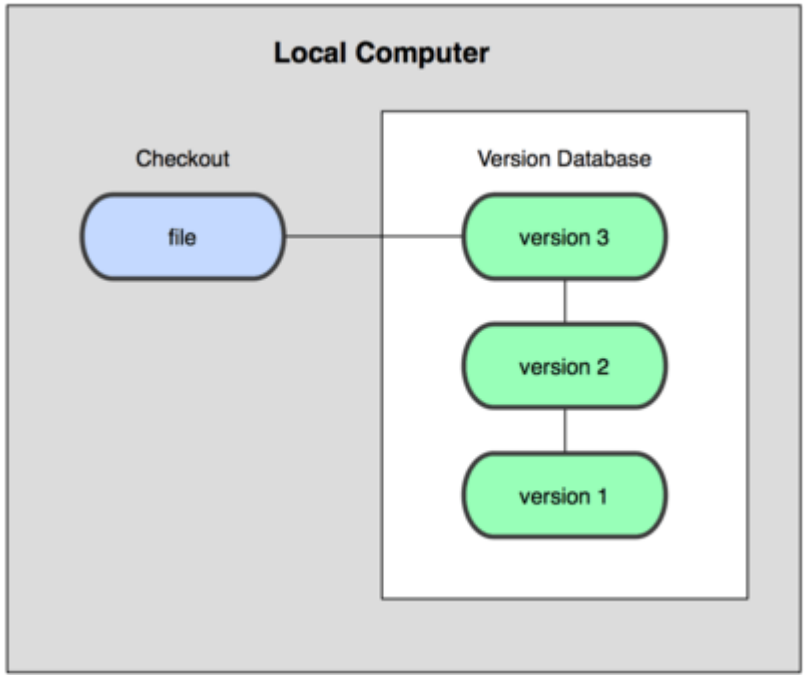
- A VCS allows you to manage file history allowing you to:

  Roll back to previous states of a file if something goes wrong

  Maintain change logs allowing you to compare versions

The simplest form of version control systems are local

- e.g. rcs in Mac OS x



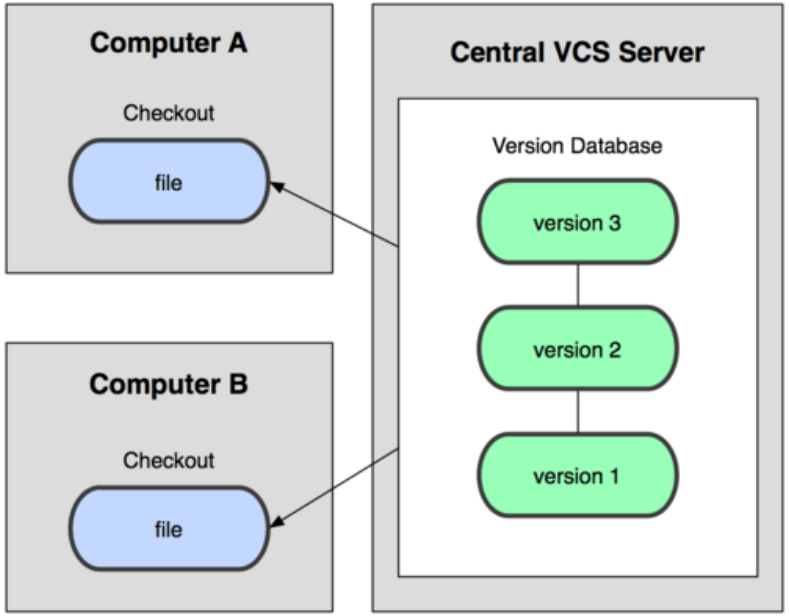
Centralised Version Control Systems

Central VCS allow multiple developers to collaborate on projects

- e.g. CVS, Subversion and Perforce

Single server that contains all versioned files

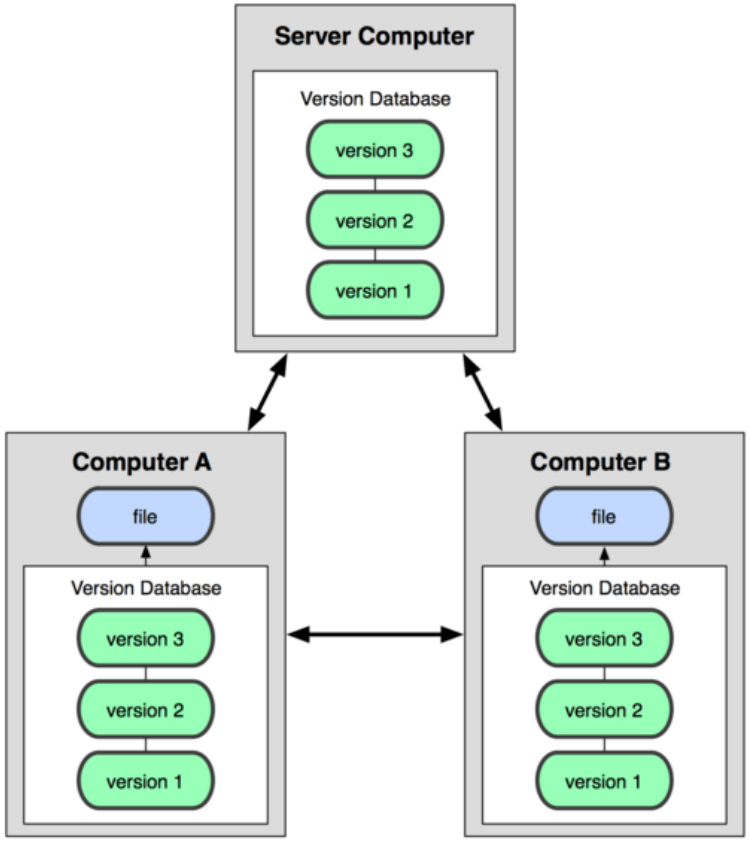
- Clients that check out files from this source



Distributed Version Control Systems

DVCS do not just check out the most local snapshot of a file

- They mirror the repository
- So each checkout is like a backup of the repository



GIT as a DVCS

GITs origin are in Linux Development and it is open source

- Its goals were to create a DVCS system that was:
  - Fast
  - Simple
  - Strong support for non-linear development
  - Fully distributed
  - Able to handle large projects like the Linux kernel efficiently

Why use source control?

Keep track of code and changes

- One copy of the code everyone has access to
- No more mailing around code and confusion trying to integrate it
- Automated version management

Allows for multiple people to edit a single project at the same time

- Push changes to the central repository
- Everyone can pull changes from others
- Merge together changes in files where there are conflicts

Branch code to work on specific parts

- Version 2.3 doesn’t need to die because someone else wants to look at version 3

Source Control using Git

Git is a popular distributed source control method

Free hosting sites available

- Bitbucket
- Github

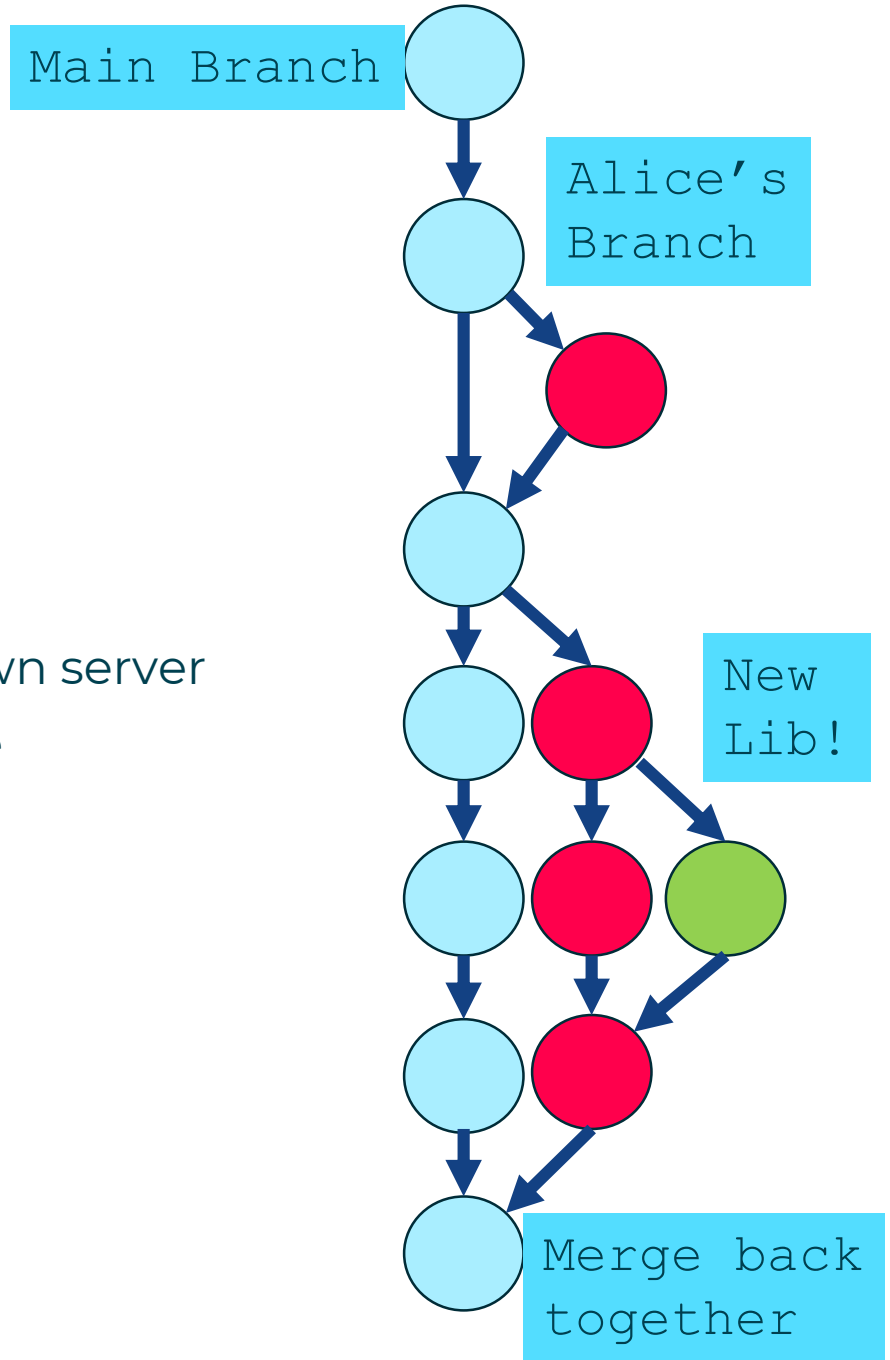
Step 1: Create a repository on a hosting site, or your own server

Step 2: Check out the repository to your own machine

Step 3: Add some code

Step 4: Commit your changes to the local repository

Step 5: Push changes to the remote repository



Demonstration – Installing GIT

You can get GIT for all major platforms:

- Linux: http://git-scm.com/download/linux
- Mac: http://git-scm.com/download/mac
- Windows: http://msysgit.github.io/

Git Configuration – Setting the User

Git keeps track of who performs version control actions

- Git must be configured with your own name and e-mail

To configure Git with your name and e-mail we use the following commands:

```
% git config --global user.name "Your Name"
% git config --global user.email "mail@example.com"
```

The values can then be accessed using:

```
% git config user.xxx
```

You can list all the configurations with:

```
% git config --list
```

ACTIVITY: GIT USER CONFIGURATION

Use the command line to configure Git with your user name and email

- Check the settings to ensure these are set

Git Commands – Etymology

Git commands all follow the same convention:

- The word ‘git’
- Followed by an optional switch,
- Followed by a Git command (mandatory),
- Followed by optional arguments

```
git [switches] <commands> [<args>]
```

```
git -p config --global user.name "Dave"
```

Getting started with Git

When you first launch Git you will be at your home directory

~ or $HOME

Through the Git Bash terminal you can then move through and modify the directory structure

Command	Explanation
ls	Current files in the directory
mkdir	Make a directory at the current location
cd	Change to the current directory
pwd	Print the current directory
rm	Remove a file (optional –r flag to remove a directory)

Activity: Git Bash - making a directory

Launch your git command line

- Ensure you are at your home directory
- Find the corresponding directory using your GUI file explorer
- Create a directory called gitTest
- Navigate within the directory using the command line
- Create a sub folder
- Navigate back to home
- Remove the gitTest directory
- Once you have completed all other tasks type history

Git Help

```
git help
```

```
git help tutorial
```

```
git help everyday
```

Activity: Git Help

Use your command line to access git help find out about:

- help
- glossary
- -a
- config
- -g

Summary

- Git is a multi-platform version control tool
- It uses a common command line interface on all major platforms
- The Git command line is based on UNIX

POWERING POTENTIAL

Any questions?

GIT REPOSITORIES

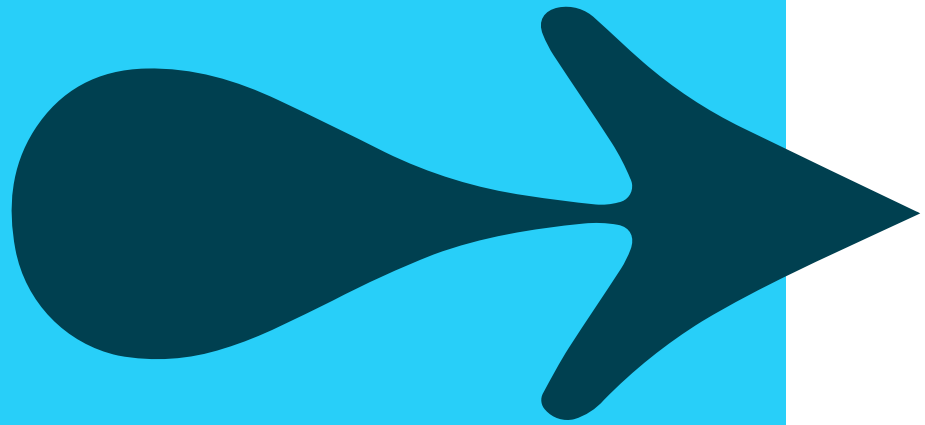
Understanding repositories

- Initialising Repos
- Adding files to repos
- Committing

Working with file staging

- How to stage and un-stage a file

Using the Git GUI in conjunction with GIT BASH



GIT Key Concepts - Repos

GIT holds assets in a repository (repo)

- A repository is a storage area for your files
- This maps to a directory or folder on your file system

These can include subdirectories and associated files

```
$mkdir firstRepo
$cd firstRepo
$git init
```

The repo requires no server but has created a series of hidden files

- Located in .git folder

Git Status

You can see the status of your git repository

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be
                                                committed)
  (use "git checkout -- <file>..." to discard changes in
                                               working
directory)

      modified:   Sample_script.sh

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

ACTIVITY: CREATING A GIT REPOSITORY

Create a folder at the ~ called firstRepo

Initialise it as a git repository

Check the status of the repo

Use the touch command to create a file called myfile.bat

Check the status of the repo

Recording Changes to a Repository

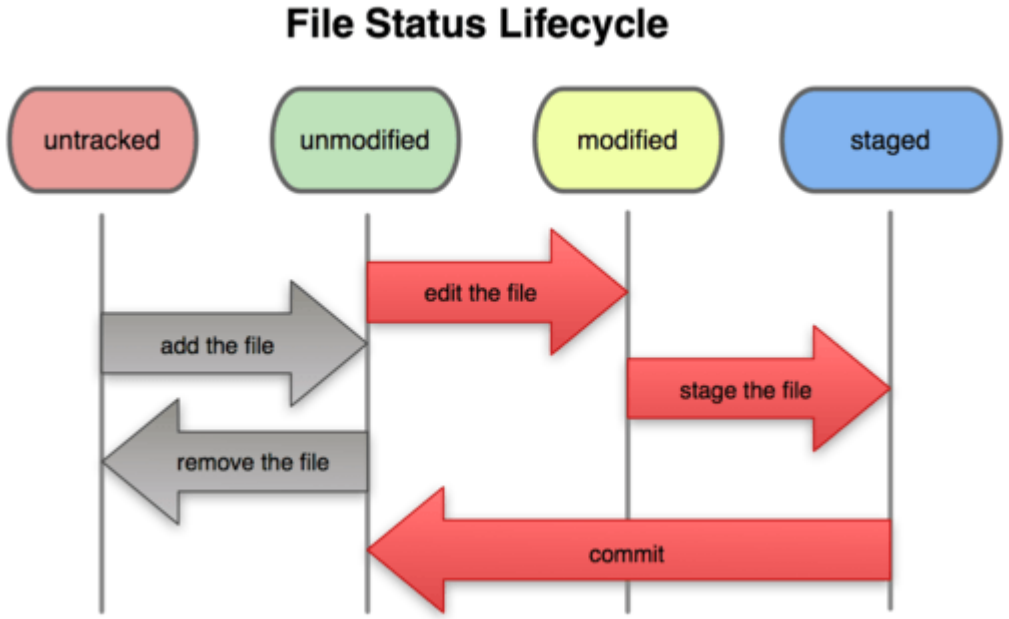
Each file in a git directory can be tracked or untracked

- Tracked files are files that were in the last snapshot

  They can be unmodified, modified or staged

- Untracked files are everything else

  Not in your last snapshot or staging area



Tracking Files in a Git Repo

To track a file we must add it to the repo

```
$ git add FILENAME
```

Then we must commit the file – adding to the repository timeline

```
$ git commit
```

This will launch the vi editor. We must then:

- Switch to input mode (i)
- Enter a commit message
- Hit ESC to complete the message
- Enter ZZ or :wq to exit the vi editor

The master repo is now updated

```
$ git status
$ git log
$ git log --stat
```

Adding files

To add a new file use the ‘add’ command

```
# a single file
$ git add specific_file_name.ext

# To add all changed files, deleted and untracked
$ git add .
$ git add --all
$ git add -a
```

Git status will show the newly added file

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

      modified:   sample_script.sh
```

Committing Files

To commit the changes use the “commit” command

- The –m flag is used to set a message

```
$ git commit -m "More content for Sample - git"
[master 93e8300] More content for Sample - git
 1 file changed, 0 insertions(+), 0 deletions(-)
```

This is now saved to the local version of your repository

GIT Key Concepts - Commits

A commit is a saved change that adds your changes to the history of your repository and assigns a commit name to it

The change is not sent to a central repository

- Other people can pull the change from you
- Or you can push the change to some other repository
- There is no automatic updating

Every commit requires a log message

- You can add a message by adding -m "your message"
- The message can be any valid string
- You can also specify multiple paragraphs by passing multiple -m options to git commit

ACTIVITY: GIT TRACKING AND COMMITTING TO A REPO

Add the myfile.bat to the tracked files of the repo

Commit it

Utilise vi to set its commit message

Check the status of the repo

Create a file called newfile.bat

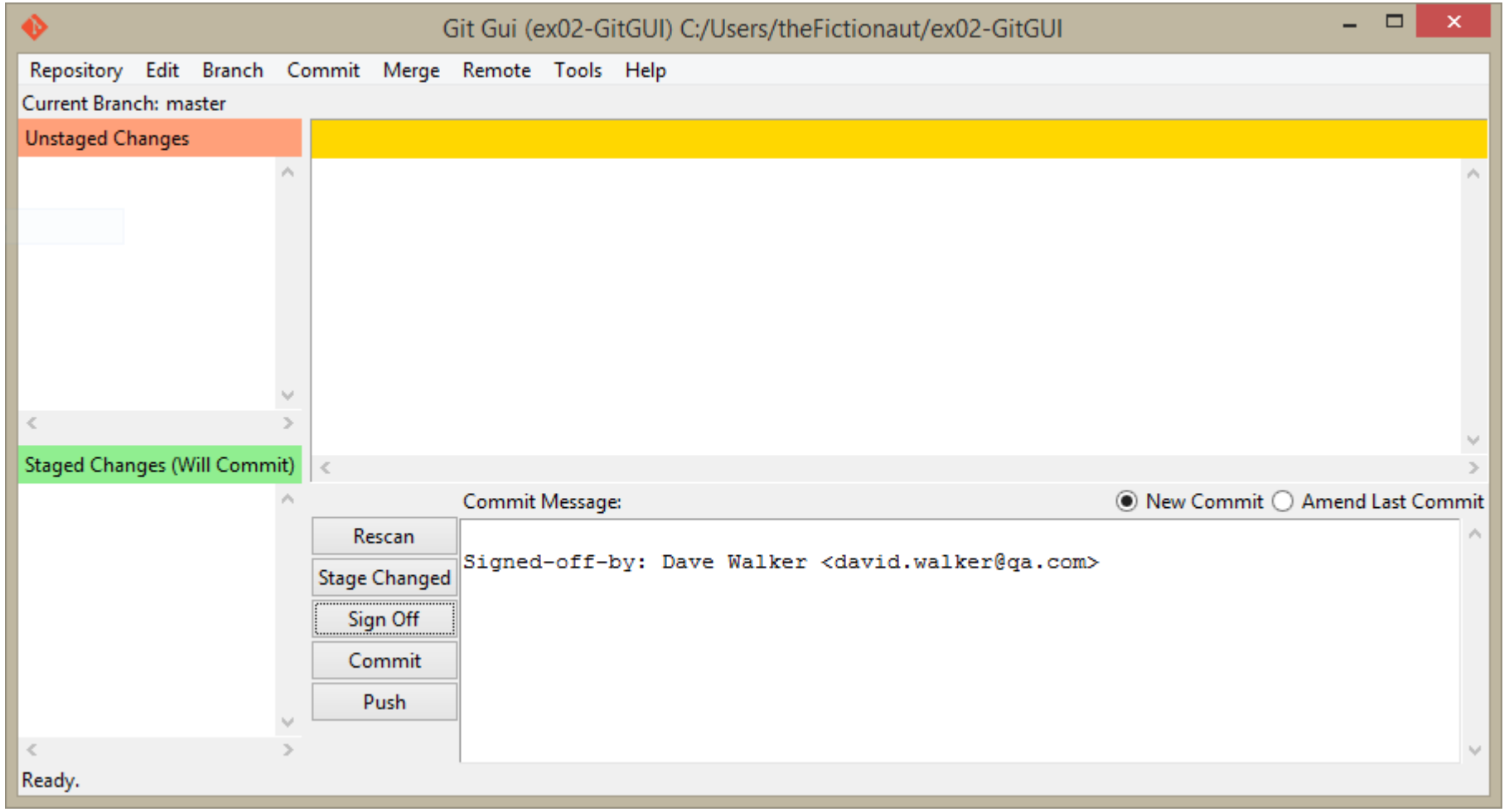
Add it to the tracked files

Check the status

Commit the file using the –m flag to add a message without using vi

Check the status using the stat flag

Demonstration – The Git GUI



WORKSHOP: WORKING WITH REPOS

Part 1 – Using Git Bash to create and stage repos

Part 2 – Using Git GUI to create and stage repos

Summary

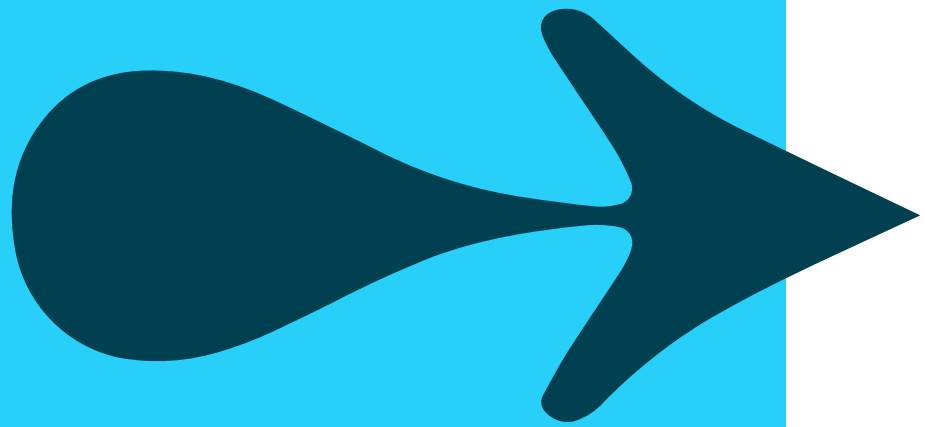
Repositories are version controlled resources that require no server

- Files within a repo can be added to the repo or ignored
- Tracked files and their status will be added to the .git resources
- Files can be worked with and committed – once committed their log history and the user status is maintained

POWERING POTENTIAL

Any questions?

UPDATES AND TRACKING



Understanding file difference and hunks

- Working with the diff command

Checking out and deleting files

- Executing Dry Runs

Git Diff

The diff command allows Git to examine a file in a repo and identify what has changed

```
$ git diff
diff --git a/cache.manifest b/cache.manifest
index 84bc776..afd8ac9 100644
--- a/cache.manifest
+++ b/cache.manifest
@@ -1 +1,2 @@
 #Comment
+"index.html"
```

Output shows the files being compared

- The string @@ -1 +1,2 @@ is known as a hunk
- Hunk shows there is a difference in the two files

ACTIVITY: GIT DIFF

Create a new directory and repo in your home directory called gitDiff

Use the echo command to create a new file called cache.manifest

- With the content:

```
# Cache Manifest v1.1
```

Add and commit the file

Use echo and the >> operator to add the content to the end of the file:

```
index.html
```

Execute a status command – note what happens

Use the diff command to understand the changes

Commit the change

Staging with Git

The Git staging area contains a version of the working directory

- That we will commit
- Git allows us to continue to make changes to staged files

Using the status command will check the working and staging areas

- The diff command has an optional --staged flag
- This can also be managed and visualised in git gui

Demonstration Staging with Git

When you add a file to a repo it is now tracked by GIT

- You can then commit the new file added to the repo

When you change a tracked file it becomes modified

- Something very interesting happens when you have multiple changes in a staged file

Demonstration:

- Using add and stage
- Examining hunk differences using diff
- Committing changes

Recording Changes to a Repository

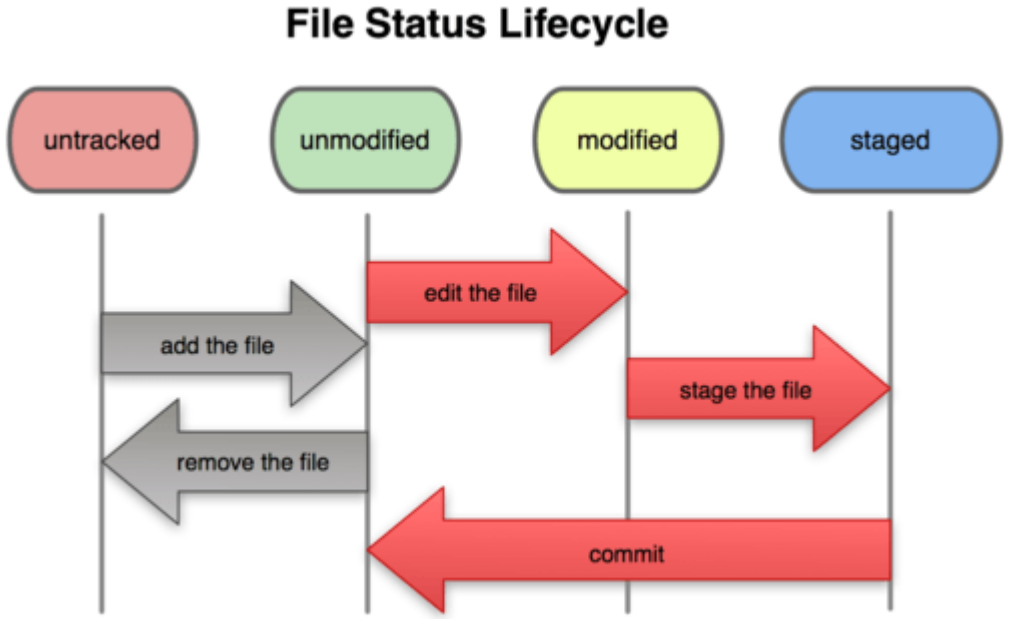
Each file in a git directory can be tracked or untracked

- Tracked files are files that were in the last snapshot

  They can be unmodified, modified or staged

- Untracked files are everything else

  Not in your last snapshot or staging area



ACTIVITY: WORKING WITH STAGING

Create a new directory called wipeout in your home folder

Initialise as a repo

Create a file called f1 and add a line of text to it

Stage the file for commit and check the status

Commit the repo

Create a second empty file with one line of text and add it to the repo

Add an additional line of text to the file

Use the diff command and note what it is telling you

Stage the file

Use the diff command again

Commit the changes

Un-staging files and reverting changes

Files once staged can be stepped backwards with a reset:

```
$ git reset <filename>
```

Changes to files can be undone with the checkout command:

```
$ git checkout <filename>
```

Untracked files can be removed with the clean command:

```
$ git clean -n
```

The reset command allows you to undo all changes since the last commit

```
$ git reset --hard
```

Adding Multiple Files to the Staging Area

Typically multiple files are added to the staging area

```
$ touch file1 file2 file3 file4
```

We might want to check the outcome of an add action is as expected

- The --dryrun flag allows you to check the outcome of an add

```
$ git add --dry-run .
```

- Then add as normal – you can view the log on a single line

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

ACTIVITY: CLEANING AND RESETTING

Within the wipeout repo add a new line of text to f1

Use diff and status to understand the current status of files

Reset the file and check the outcome via diff and status

Create files f2,f3,f4 – add a line of text to f2

Add f2 to the repo

Use Status to understand the changes

Use reset to undo the changes in this lab

Use clean to remove the untracked files

Deleting and Renaming Files in Git

We use the rm command in git to remove files

- Then run status to check

```
$ git rm file1
```

```
$ git commit -m "file1 removed"
```

Renaming can be achieved with the mv command

```
$ git mv file new_name
```

Tracking Files in a Git Repo

To track a file we must add it to the repo

```
$ git add FILENAME
```

Then we must commit the file – adding to the repository timeline

```
$ git commit
```

This will launch the vi editor. We must then:

- Switch to input mode (i)
- Enter a commit message
- Hit ESC to complete the message
- Enter ZZ to exit the vi editor

The master repo is now updated

```
$ git status
$ git log
$ git log --stat
```

When to Commit?

Commit little and often when using GIT

It is normal to commit when:

- Adding or deleting a file
- Renaming a file
- Updating a file known to be in a good working state
- When you anticipate being away from work
- When you introduce ‘questionable’ code

GIT Key Concepts - Commits

A commit is a saved change that adds your changes to the history of your repository and assigns a commit name to it

The change is not sent to a central repository

- Other people can pull the change from you
- Or you can push the change to some other repository
- There is no automatic updating

Every commit requires a log message

- You can add a message by adding -m "your message"
- The message can be any valid string
- You can also specify multiple paragraphs by passing multiple -m options to git commit

ACTIVITY: GIT TRACKING AND COMMITTING TO A REPO

Add the myfile.bat to the tracked files of the repo

Commit it

Utilise vi to set its commit message

Check the status of the repo

Create a file called newfile.bat

Add it to the tracked files

Check the status

Commit the file using the –m flag to add a message without using vi

Check the status using the stat flag

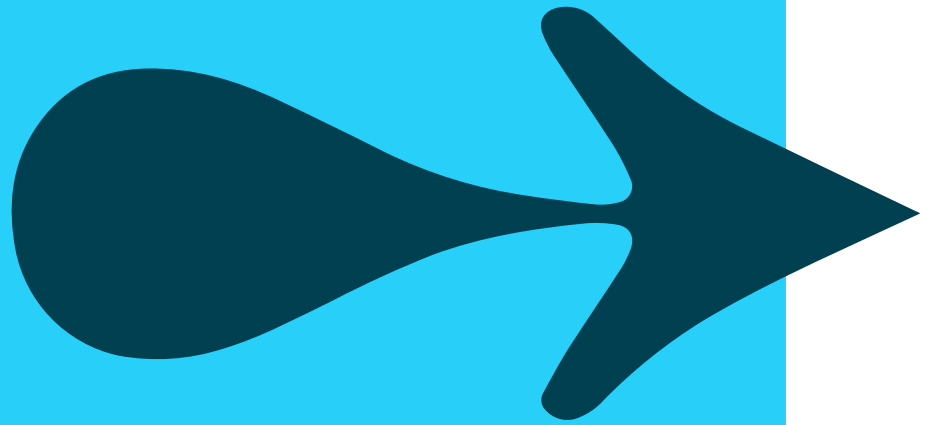
Summary

- Git diff
- Un-staging
- Dry runs
- Deleting files

POWERING POTENTIAL

Any questions?

BRANCHING



Logging a – deeper dive

- Patch and Tree views

How branches work

- The HEAD and moving it

Creating Branches

- Managing and switching branches
- Committing changes to branches
- Merging Branches

Git Logging – A Deeper Dive

When we use git log we see a detailed history of the history log

- Provided in reverse chronological order

Each entry in the log has a SHA1 ID making each commit unique

- This is a cryptographically strong ID

The commit also holds author and date meta information

- Plus a log message

Each commit always points back to its parent

- We can visualise the history with gitk

GITK - Patch and Tree views

The GITK viewer has two ways to see the changes:

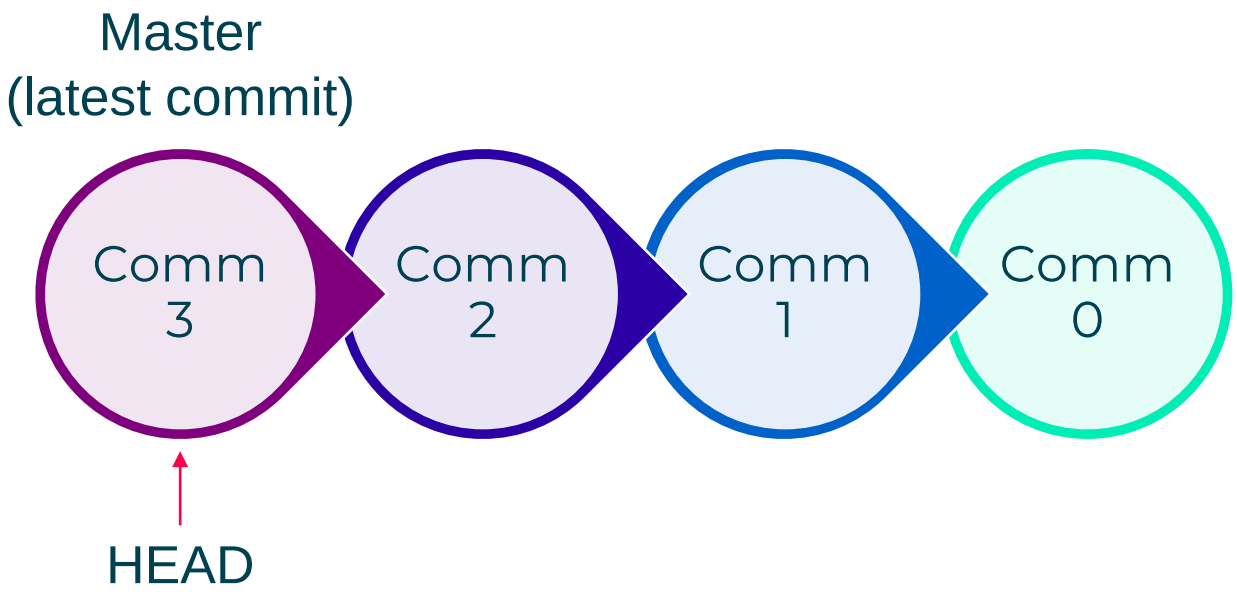
- The **Patch** view show the files that were patched in a commit
  Showing how the files were changed
  Shown as a flat directory reflecting the changes

- The **Tree** view shows the files changed and the meta data
  It represents the directory structure of the project
  But not the details of the hunks

How Branch History Works

Commits occur in a line created when we first initialise the repo

- Git also creates a default Branch called Master
- Master is a pointer to the last commit in the branch
- The HEAD represents our current location
- Normally in the same place as Master



Git Checkout

The HEAD can move – checking out previous commits

- You need the SHA1 id to achieve this – use log to find it

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

Then use the first 4 characters of the ID to checkout what you want

```
$ git checkout YOUR_SHA1ID
```

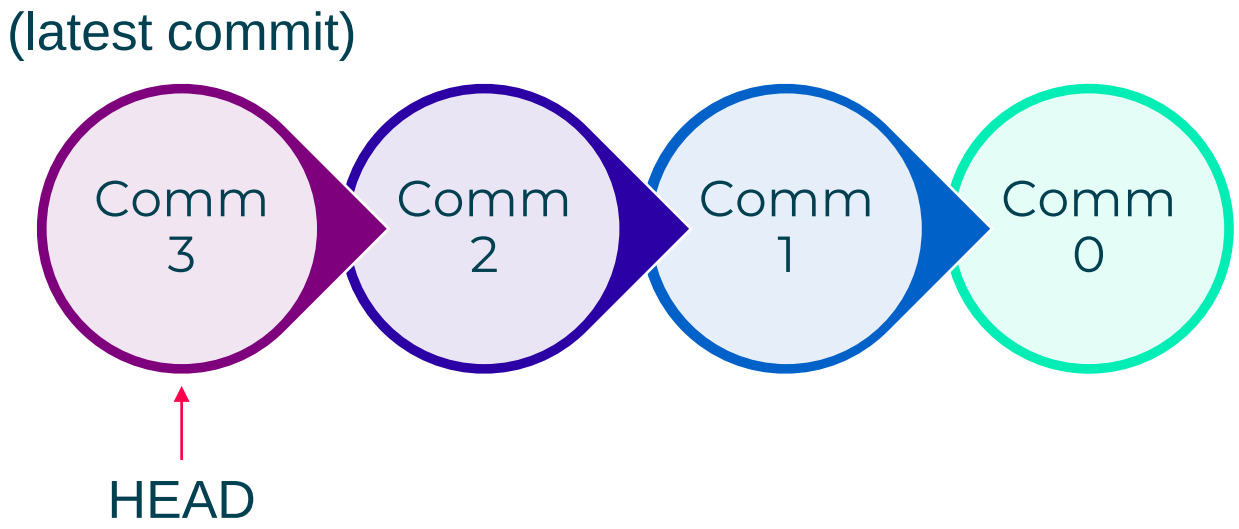
```
$ git checkout 5398
Note: checking out '5398'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

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

  git checkout -b new_branch_name
```

Detaching the HEAD

Normally the HEAD is attached to a branch but we can move it

- This is what we did with checkout

When we move the HEAD from the Master, git considers it detached



ACTIVITY: WORKING WITH THE HEAD

In your most recent repo use the one line log to locate the sha1 history

Move the HEAD to the start of the commit history via checkout

Check the log history

Use touch to create a new file

Add it to the repo

Commit the changes

Use checkout to return to the master

Check the contents of your directory and note what has happened to the new file

Check the log history

Tagging

Tagging allows you to add metadata to a repository

- Often used for versioning

```
$ git tag
v0.1
v1.3
```

You can search for tag patterns using the –l flag

```
$ git tag -l 'v1.4.2.*'
v1.4.2.1
v1.4.2.2
```

You can create an annotated tag with the –a flag

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

ACTIVITY: GIT TAGGING

Re run your commands from the previous activity

Create a suitable tag on your first commit

```
$ git tag where_it_all_began -m "Committing a tag at the start"
```

Move the HEAD back to the Master

Run gitk to explore the tag

Use the one line log to find a different SHA1 id

Create another tag (no need to move the head this time)

Check the outcome in gitk

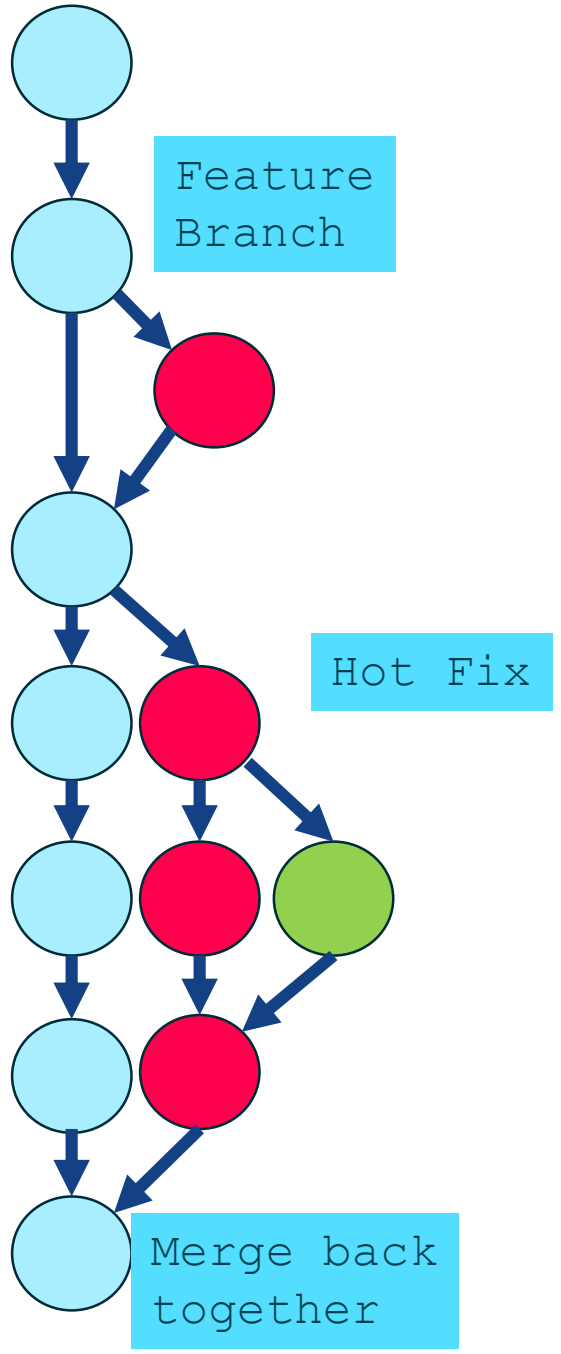
Use git show and the tag name in bash to see the diff history

Checkout the second tag you created

Git Key Concepts - Branchi

Branches are other “paths” or “lines” of development

- Imagine that you’re making a website for a client, and it consists of an HTML file and a CSS file. You work on these two files in a default repository. This default repository is also conveniently associated with a default branch, which is often called “main”, or “trunk”.



Why and When to Branch – New Features

The master branch often represents the working code base

- i.e. IT WORKS!
- So we don’t hack on the master

Branching is where we can safely introduce new code

- You can work and test in the branch and only merge once you are ready

Why and When to Branch – Hot Fixes

Branches isolate changes from the code base

- You can navigate to any prior commit and check it out
- From that point, create a new branch
- You can then work on the fix and merge when ready

```
$ git checkout -b hotfix thepast
```

The above code creates a new branch from a previously defined start point

- The start point can be a branch, tag or SHA1 ID

Git Branching

Branching allows you to diverge from the main line of development

- Without doing accidental damage to the main line

Git branches are very lightweight compared to other version control systems

- Git encourages a workflow that allows you to branch and merge

Branches build on core Git features

- When you commit you have a snapshot of current content
- Plus zero or more pointers to the current commits

  Based on this repository or a parent repository

```
$ git branch testing
```

Branches

Branches allow for teams to add new features and work in parallel without adding ‘dangerous’ code and changes to the main branch of the repository

- The trunk (main) should be kept error free

```
# create a branch
$ git branch [name]

# list all the branches
$ git branch --list

# delete a branch
$ git branch -d [name]

# switch to a branch
$ git checkout [name]

# push to branch on remote
$ git push --set-upstream origin
```

Merging Branches - Types

There are two types of merge

- Fast Forward

  If there are no changes on the master that will conflict then the head is updated to the end point of the branch

- 3 way merge

  A new point is created with the two branches merged together

```
# switch back to the master branch
$ git checkout master

# merge the branch
$ git merge [branch name]
```

Merging Branches

Git merges the traffic *into* the current branch

- So make sure you are on the correct branch!

```
$ git merge <branchname>
```

The diff command can be used to explore the differences between branches

```
$ git diff <master>...<branch2>
```

The output shows the difference between branches

- The order of the output matters – the master is shown first
- Showing that branch2 will be merged into master

Demonstration: Branches, committing and checking out

In this demonstration we will

- Do work on a website
- Create a branch
- Work in the branch

We will then do a hotfix

- Switch back to the production branch
- Create a branch to add the hotfix
- Test and merge the branch
- Switch back to the original story

ACTIVITY: MAKING AND DELETING A BRANCH

Create a new directory called DevOps off your home directory

Initialise it as a git repository

Add a file called bar and commit it

Use the branch command to see the current branches

Create a new branch called ops and check it out

Delete the master branch

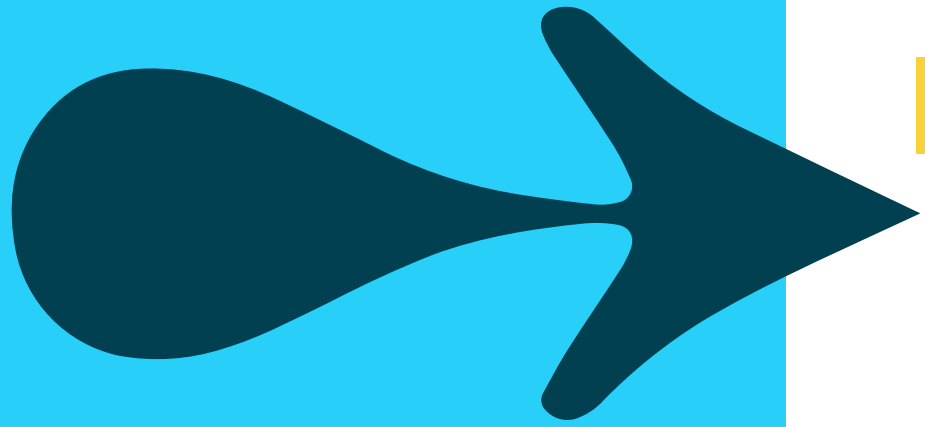
Summary

- **Git is a time machine – your history is maintained in an indexed list**
- Each entry has a SHA1 key
- **You can branch from your code allowing you to fix and edit code**

POWERING POTENTIAL

Any questions?

CLONING



Cloning makes a physical copy of a Git repository

- It can be done locally or via a remote server e.g. GitHub
- You can push and pull updates from the repository

The benefit of cloning repos is that the commit history is maintained

- Changes can be sent back between the original and the clone

Cloning is achieved with the clone command:

- Or through the GUI

```
$git clone source destination_url
```

The GUI branch visualizer gives us a very useful way to see the origins of branches

Git on the Server

You will either need to create or access a Git Server

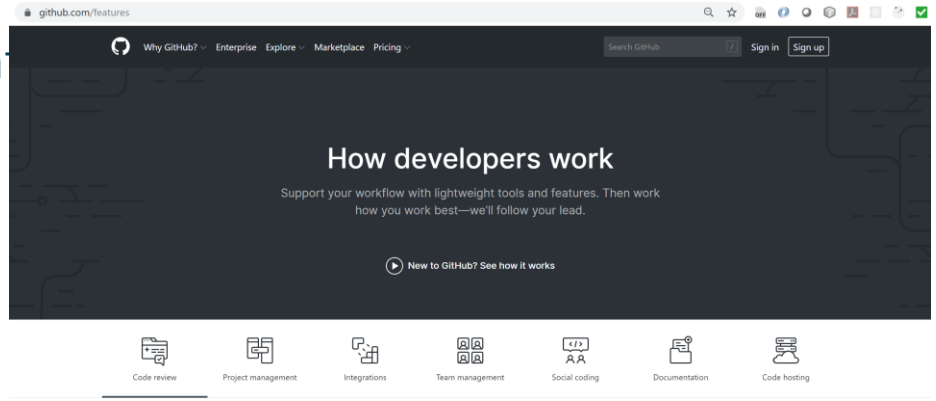
When getting started with Git a hosted solution is a grea

- GitHub is the most common of these

  User centric solution

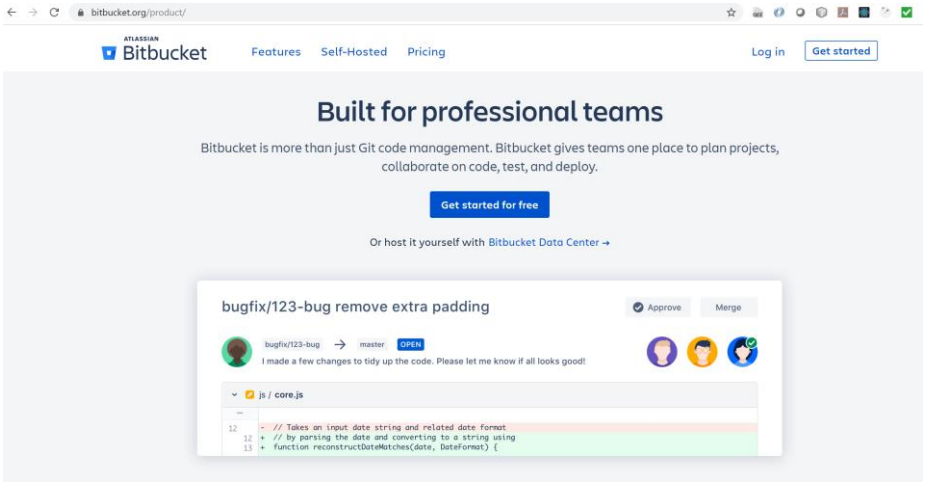
  Uses SSH protocol or HTTPS

  https://github.com



- Bitbucket is also popular

  https://bitbucket.org



Cloning a Repository

Cloning copies the entire repository to your hard drive

- The full commit history is maintained

To clone a remote repository

```
$ git clone [repository]

$ git clone https://username@bitbucket.org/username
/repositoryname
```

To clone a specific branch

```
$ git clone -b branchName repositoryAddress
```

Demonstration – Cloning and Visualising Using the GUI

Working with Remote Repositories

Git projects are often held on remote repositories

- These hold versions of a project or dependencies on the web/network

To see configured remote repositories run the git remote command

- If you have cloned a repository you should see the origin
- To add a repository:

```
git remote add [shortname] [url]
```

- shortname becomes an alias for access to the repository

When you have your project at a point you want to share you have to push it upstream

```
$ git push origin master
```

Pushing to the Repository

To update the changes in the remote repository use the ‘push’ command

```
$ git push origin master
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 6.81 KiB | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To https://username@bitbucket.org/username/demo.git
   03d97e5..93e8300  master -> master
```

To set the branch to automatically push to use

```
$ git push --set-upstream origin master
```

Pulling the Repository

To pull all the changes made to the repository then we can use the ‘pull’ command

```
$ git pull
```

It Is good practice to pull the repository before pushing changes

- You get an up to date copy of the repo to push to
- You can see any conflicts before they are pushed
- You can stash your changes before pulling the remote branch

Stash and Merge

Stash saves your changes locally, allowing you to pull the latest copy of the repository without overwriting your changes

- The changes are stored in a stack
- You can see all the stashes with git stash list

```
# to stash your changes
$ git stash

# list all the stashes
$ git stash list

# pull the repository
$ git pull

# pop the top entry off the stash and merge
# with the changes
$ git stash pop
```

Stashing Branches

Stashing allows us to pause some work on existing files

- Without needing to make a full commit
- This gives you a clean working directory so you can commit

```
$ git checkout -b newbranch
$ vi file //make a change
$ git stash
$ git checkout master
```

Popping the Stash

The stash stores work in progress (WIP) – you can return to using the pop command

```
$ git stash list
```

```
$ git stash pop
```

ACTIVITY: WORKING WITH THE STASH

Create a new directory and repo called stashing

Use vi to create a new file called mywip and add a line to it

Add the file then commit the branch

Modify the file again and commit the branch

Use the log to find the original SHA1 key

Move the head to that location and create a new branch

Modify mywip and save

Try to switch to the master branch – what happens?

Stash the current wip – what happens

Now switch to the master branch

Check the status of the stash

Pop the most recent sashed operation

Change the file and commit

Switch to the master

Resources

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

https://git-scm.com/docs/user-manual.html