TopHat activities

- Please join and indicate that you have done so:

- https://app.tophat.com/login
  Join: **147651**

UNIVERSITY OF LEICESTER

BASICS OF GIT VERSION CONTROL

CO1106 Requirements Engineering and **Professional Practice**

Dr Matthias Heintz , Prof. Shigang Yue
(mmh21@leicester.ac.uk) (sy237@Leicester.ac.uk)

Introduction to part 2 (Weeks 6-11)

- The first part of CO1106 covered the process of gathering requirements and you learnt how and why it is an integral part of Software Engineering (SE)
- In the second part of the module, we'll be looking at different aspects of 'Professional Practice', whilst continuing the projects from the first part

But what is professional practice?

- It is the '**conduct and work of someone in a particular profession**'
- It embodies the **knowledge**, **skills** and **attitudes** that workers must possess in order to conduct their work in a professional, responsible and ethical manner

Why are we covering git in this module?

- There are two parts to the module title:

  - Git comes under **professional practice**...
  - Many companies that require employees to share code will use Git...
    - Therefore, knowing how to use it will benefit you

Git & the Linux Command Line Interface

- 'Git' is a command line tool that you'll be **using to submit for CW2** (via 'GitLab')
- Here, we assume some experience of the Linux CLI (as per CO1101)
- You should try and refresh your memory of basic commands such as:

'`cd ...`' (for changing directories)	'`ls ...`' (list files in current directory)
'`mkdir ...`' (makes a new directory)	'`mv ...`' (rename/move files)

- The following is a good reference: Linux CLI cheat sheet (by Dave Child) (https://cheatography.com/davechild/cheat-sheets/linux-command-line/)
- **Please also watch the 'Linux CLI refresher' available on Blackboard!**

Schedule

Week	Start Date	Monday / Wednesday - Lecture	Thursday / Friday - Surgery	Assessment
26	15/01/2024	Introduction & Why Requirements?	Icebreaker activity for groups & work on Project Description	
27	22/01/2024	Requirements gathering (Quan. & Qual. User Studies)	Work on requirements gathering for Assessment 1	
28	29/01/2024	Functional Requirements	Work on building list of funct. requirements for Assessment 1	
29	05/02/2024	Non-Functional Requirements	Work on building list of non-funct. requirements for Assessment 1	
30	12/02/2024	Overview of UML; Use Case diagrams and descriptions	Work on Use Case diagram and Use Case description for Assessment 1	
31	19/02/2024	Basics of git version control	Checkout and setup group git repository and set up Weekly Log .md file	Assessment 1 (50%)
32	26/02/2024	More advanced git topics	Work on reworked list of functional requirements	
33	04/03/2024	Class Diagrams	Work on Class diagram	
34	11/03/2024	Class Modelling	Rework Class diagram	
35	18/03/2024	Sketching and Lo-fi prototyping	Work on wireframes/lo-fi prototypes	
36	25/03/2024	Software Laws & Professionalism	**none**	Effective use of Git (10%)
37-40	01/04/2024	break	break	
41	29/04/2024	none	none	Blackboard Test (40%)

Matthias
Shigang

Session objectives

- At the end of the lecture you will be able to:
  - *Explain what ‘source control’ is*
  - *Explain its benefits and why it is so widely used by professionals in a wide variety of programming-based disciplines*
  - *Be able to effectively use (basic features of) the ‘git’ source control platform*

https://app.tophat.com/login
Join: **147651**

- Who knows what '**source control**' is?

- Who has worked with (or is aware of) '**git**'?

Group coursework in CO1106

- Main group project
  - ***Part 1 (50% - due 23rd February)***
    - Project description (10%)
    - Quantitative and qualitative studies (10%)
    - Written requirements (20%)
    - Use Case UML Diagram and Use Case Description (10%)
  - ***Part 2 (10% - due 27th March)***
    - Effective usage of git version control

Group Coursework Part 2 – Effective use of git version control

- Your group will utilise a git repository in order to manage/submit any files produced as part of the second part of the group project for CO1106 (details of how to access the repository will be provided to you in the tutorial of Week 6).
- Groups should make frequent usage of their group repository - any shared files that you work on (for example, the .md files containing functional requirements and use case descriptions) should be added to the repository as soon as they are made, with regular changes being committed by group members until that particular file is finished. Each group will be responsible for coordinating their git usage.

Group Coursework Part 2 – Instructions

- A maximum of 3 marks are available, depending on how effectively your group used git. We will decide the number of marks you receive by inspecting the contents of your repository as well as the commit history of your repository:
  - For 1 mark, at least one member of your group needs to make a commit each week; no advanced features (i.e., branching/merging) have been used, and commit descriptions (included when you made the commit) may be non-descriptive and not give a good idea of the changes included in a particular commit.
  - For 2 marks, multiple commits should be made each week, and an initial (possibly incomplete) version of the artefact worked on during each tutorial session needs to be submitted in the week it was worked on. Commit messages must be descriptive (but succinct) and give a good idea of the changes that have been committed.
  - For 3 marks, you must satisfy all points from the previous two bullets, and it should be apparent from your commit log that all members of the group have

Group Coursework Part 2 – Weekly log

- Each group must also produce **a 'Weekly Log' (.md format)** that is updated with the following contents in each of the weeks 6-10 (5 weeks in total):
  - A 'beginning of week' entry containing a summary of the work that the group plans to complete during that week, along with a breakdown of which members will complete which tasks. The beginning of week entry for each week should be produced by the group; for example, during your first groupwork meeting of that week.
  - An 'end of week' entry which lists the tasks that each member of the group has completed; any outstanding work; and any other additional information that your group feel is relevant to add .

Group Coursework Part 2 – Weekly log (continued)

- Each group will be responsible for designing the Markdown structure of their Weekly Log, ensuring that it is easy to read and maintained properly. The entries in the Weekly Log will be checked on a weekly basis (the entry for Week X will be checked by us during Week X+1). For the entries of Week X, there is a maximum of 0.75 marks available (up to a total of 3 marks for weekly updates):
  - 0.25 marks depending on whether both the 'beginning of week' and 'end of week' entries have actually been added to the log (if either one is missing, you receive 0 marks for that week)
  - 0.5 marks will be awarded depending on the quality of the entry (is it descriptive enough? does it contain all the information listed above?)
  - An additional mark out of 4 will be awarded based on the readability/quality of the Weekly Log document.

Marking rubrics

Markdown Usage, Marking Rubric					
Fail	Poor	Requires Improvements	Satisfactory	Good	Excellent
0	0.5	1	2	3	4
No serious attempt is made.	Required information is contained in the document but no Markdown syntax has been utilised in order to improve readability.	A limited amount of basic Markdown syntax has been used to produce a document that is readable but not aesthetically pleasing or easy to navigate. There may be some syntactical errors in the Markdown usage that cause the document to look untidy.	Basic Markdown syntax has been used to reasonable effect. The produced document is navigable and its contents is clearly displayed. Most audiences should have little to no problem with understanding the document, but it is not particularly aesthetically pleasing.	Same as Satisfactory, but with a number of more advanced Markdown features used in order to improve the readability of the document. Any member of the intended audience would be able to navigate the document easily.	Same as Good, but a clear effort to make the document aesthetically pleasing as well as easily navigable has been made.

- **GitLab** Repositories **will be set up for each group**; before this week's tutorial sessions, where you will clone them for the first time.
- You'll be required to commit a number of components (Class Diagrams, Sketches, Prototypes) to your group repository, including a 'Weekly Log'
- To create the weekly logs and other text-based files, you'll be using a very simple markup language called **Markdown**

GITLAB

GitLab

- GitLab is similar to GitHub; it provides web-hosted git repositories, but it is aimed more towards enterprise and large-scale business scenarios
- The University uses GitLab to provide repository hosting for group projects and BSc/MSc final year projects
- To access your University GitLab account, go to: https://campus.cs.le.ac.uk/gitlab/users/sign_in and enter your IT username (no email extension) and ***LINUX PASSWORD***

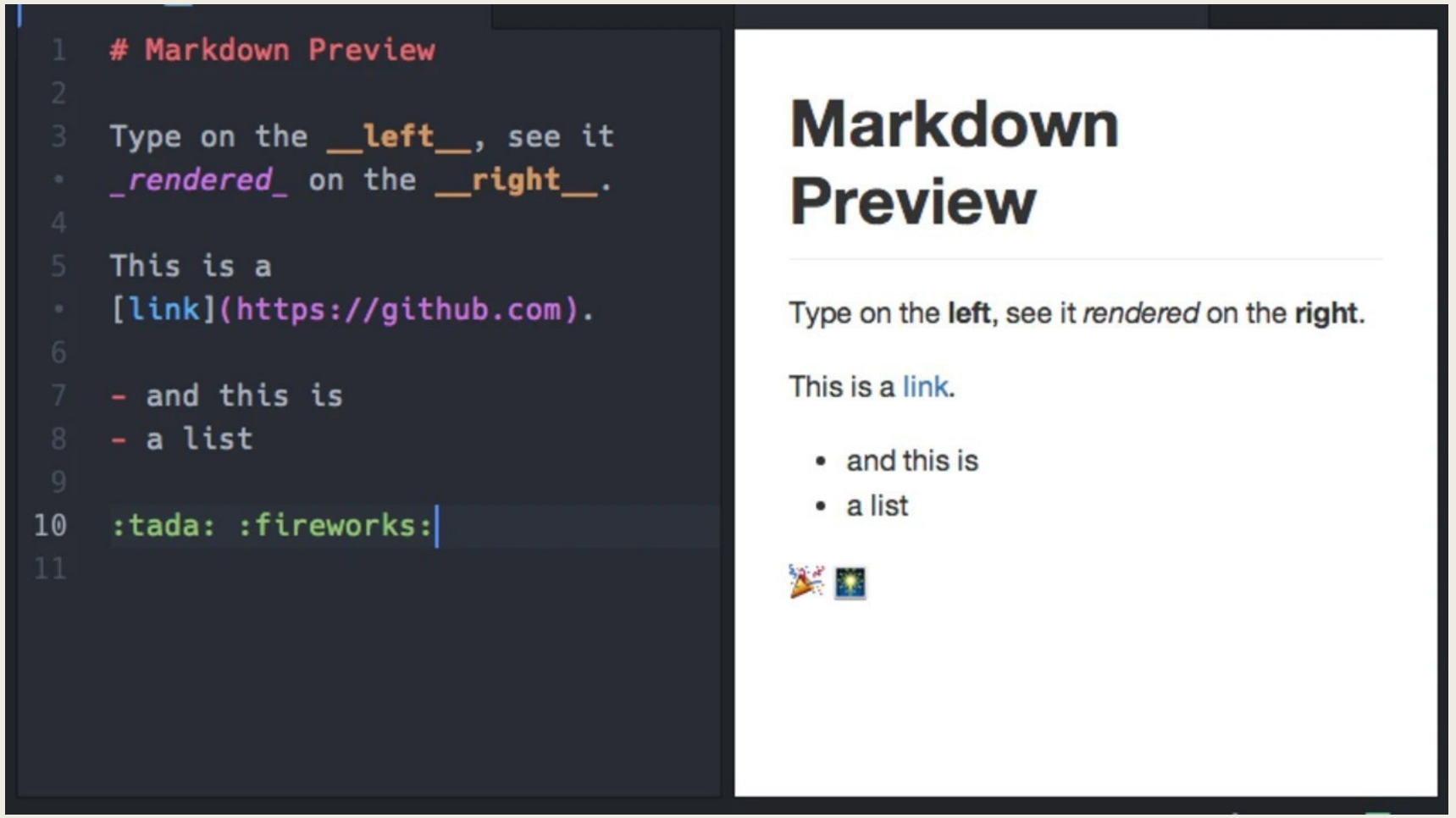
- **In this week's tutorial you'll follow a guide to get you started out in GitLab**

MARKDOWN

Markdown

- **Markdown** is a lightweight markup language
- Widely used to create simple webpages and documentation for code/projects stored in hosted repositories (e.g. on GitHub or GitLab)
- Markdown syntax is **very simple** and there a variety of web-based editors that can be used to create **.md** files
- Designed so that the plaintext version of an **.md** document is human-readable

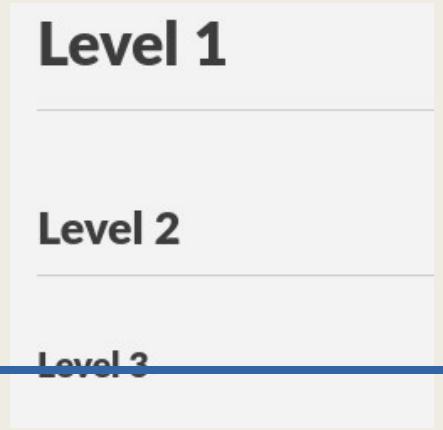
Markdown example



Some basic Markdown syntax

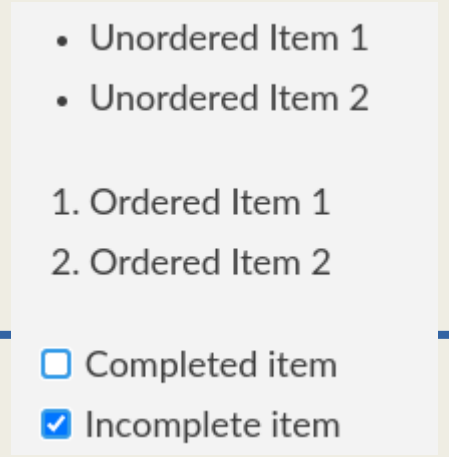
Headings

```
# Level 1

## Level 2

### Level 3
```



Lists

```
- Unordered Item 1
- Unordered Item 2
1. Ordered Item 1
2. Ordered Item 2
- [ ] Completed item
- [x] Incomplete item
```



Complete cheatsheet available on Blackboard...

Text modifications

```
**Asterisks make text bold** and    so do
double underscores
==Double equal signs highlight text==
*Asterisks italicise text* and so do _single
underscores_
```

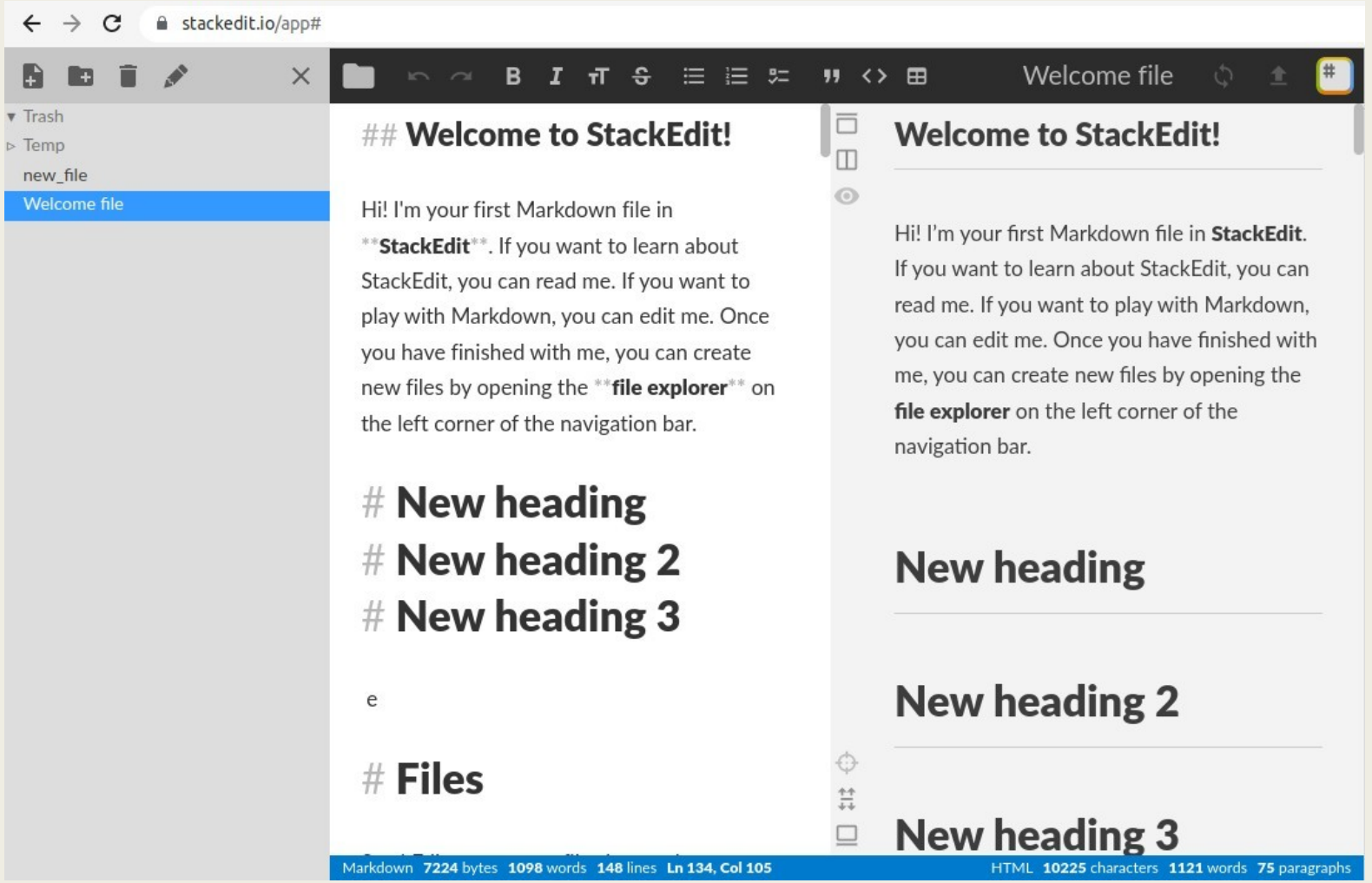
Asterisks make text bold and **so do double underscores**

Double equal signs highlight text

Asterisks italicise text and so do *single underscores*

A web-based .md editor: StackEdit

- There are various editors that you can use…
  - *'StackEdit'*
  - *'Dillinger'*
  - *Any standard text editor*



SOURCE CONTROL

Source control: What is it and what's the point?

- Source/version control is the practice of **tracking and managing changes to collections of information** (e.g., a folder structure on your PC) using a **VCS** (version control system, e.g. 'git')
- A **VCS** (Version Control System) equips a folder with a special kind of database (called a **repository**) which stores all previous versions of files contained in that folder
- This enables us to **revert to an older version of a file** if we make mistakes (i.e., we overwrite work in a file unintentionally)
- It also allow groups of developers to **work on a common project**, whilst **reducing the likelihood** of update conflicts (we'll see how later...)

Sounds useful! 

‘Git’ for local file history and backup

Why is git useful? An example

A student is working on her coding assignment late at night and decides she doesn’t like the way she’s written a certain function...

Why is git useful? An example

A student is working on her coding assignment late at night and decides she doesn’t like the way she’s written a certain function...

She has a new idea of how to write it and begins to rework the code.

Why is git useful? An example

A student is working on her coding assignment late at night and decides she doesn’t like the way she’s written a certain function...

She has a new idea of how to write it and begins to rework the code.

She goes to sleep and plans to finish the changes in the morning...

Why is git useful? An example

A student is working on her coding assignment late at night and decides she doesn’t like the way she’s written a certain function...

isn t

She has a new idea of how to write it and begins to rework the code.

She goes to sleep and plans to finish the changes in the morning...

When she wakes up, she realised that the new method correct so she wants to use the old, working version of the function.

Why is git useful? An example

A student is working on her coding assignment late at night and decides she doesn’t like the way she’s written a certain function...

isn t

She has a new idea of how to write it and begins to rework the code.

She goes to sleep and plans to finish the changes in the morning...

When she wakes up, she realised that the new method correct so she wants to use the old, working version of the function.

Why is git useful? An example

A student is working on her coding assignment late at night and decides she doesn't like the way she's written a certain function...

isn't

She has a new idea of how to write it and begins to rework the code.

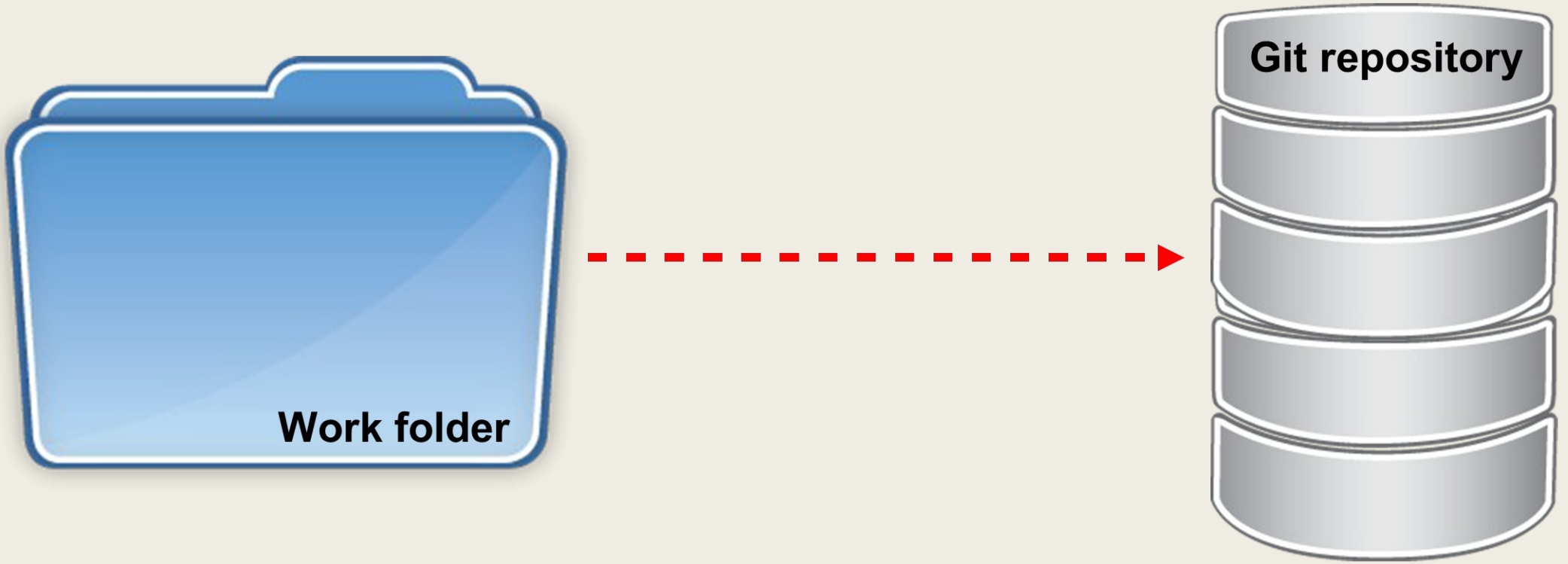
She goes to sleep and plans to finish the changes in the morning...

When she wakes up, she realised that the new method correct so she wants to use the old, working version of the function.

But, she's already overwritten the previous version! Now she has to write it again from scratch… If only she had **backed up her work somehow.**

If she'd been using Git version control from the start, resolving this would've been easy...

Git allows us to make backups or ‘saves’ (called **‘commits’**) of our work that we can easily revert to if a problem occurs…

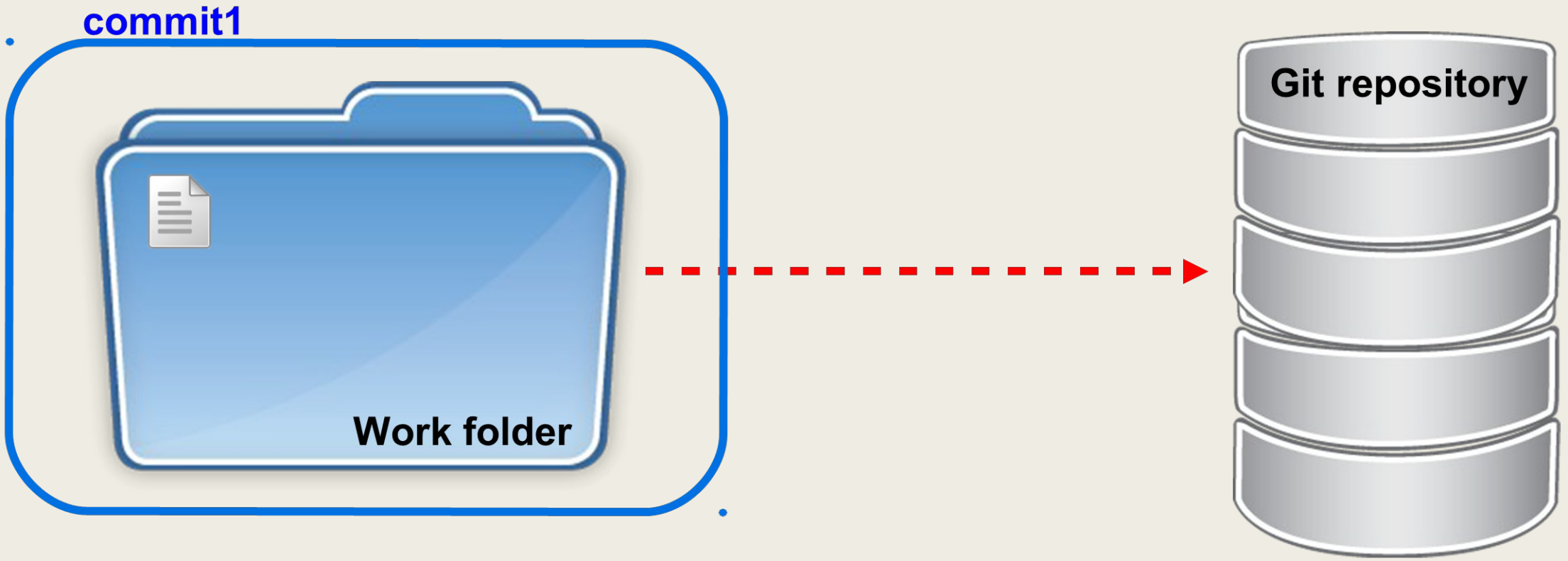


As we add/delete/edit files in our work folder, we can tell git to store a new ‘snapshot’ of our folders current state (a bit like saving a computer game)

Git allows us to make backups or ‘saves’ (called **‘commits’**) of our work that we can easily revert to if a problem occurs…

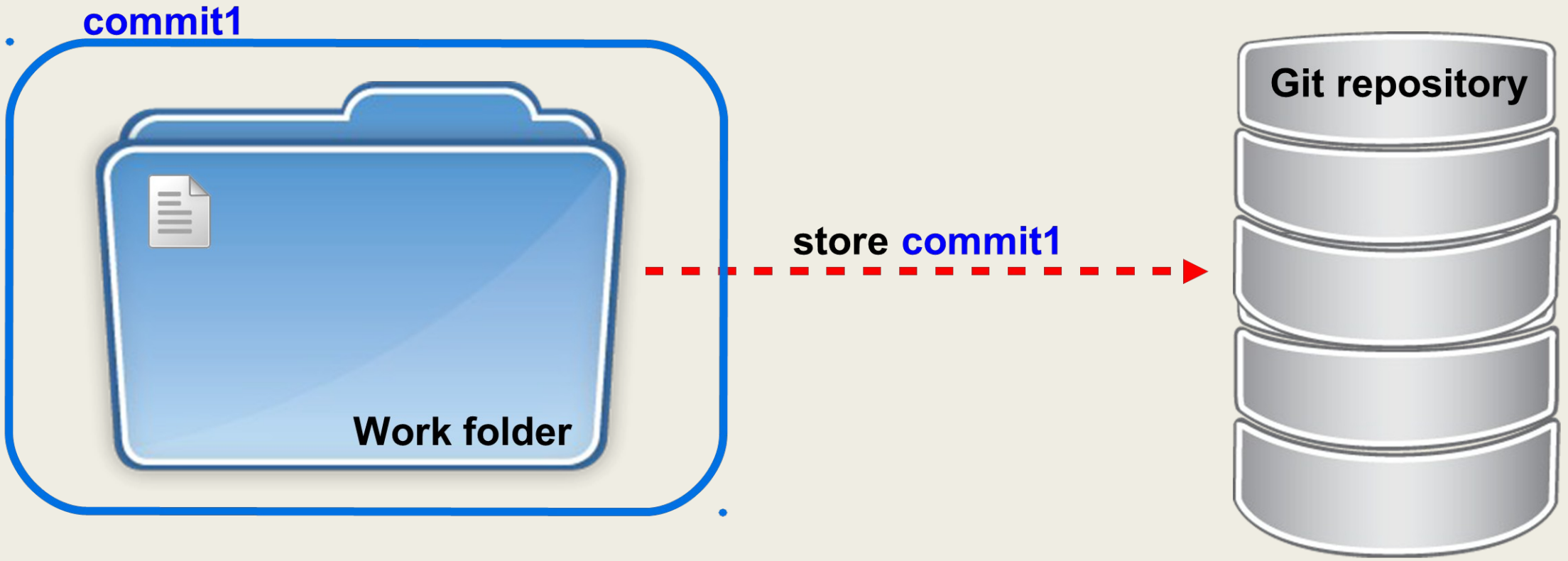
As we add/delete/edit files in our work folder, we can tell git to store a new ‘snapshot’ of our folders current state (a bit like saving a computer game)

Git allows us to make backups or ‘saves’ (called **‘commits’**) of our work that we can easily revert to if a problem occurs…



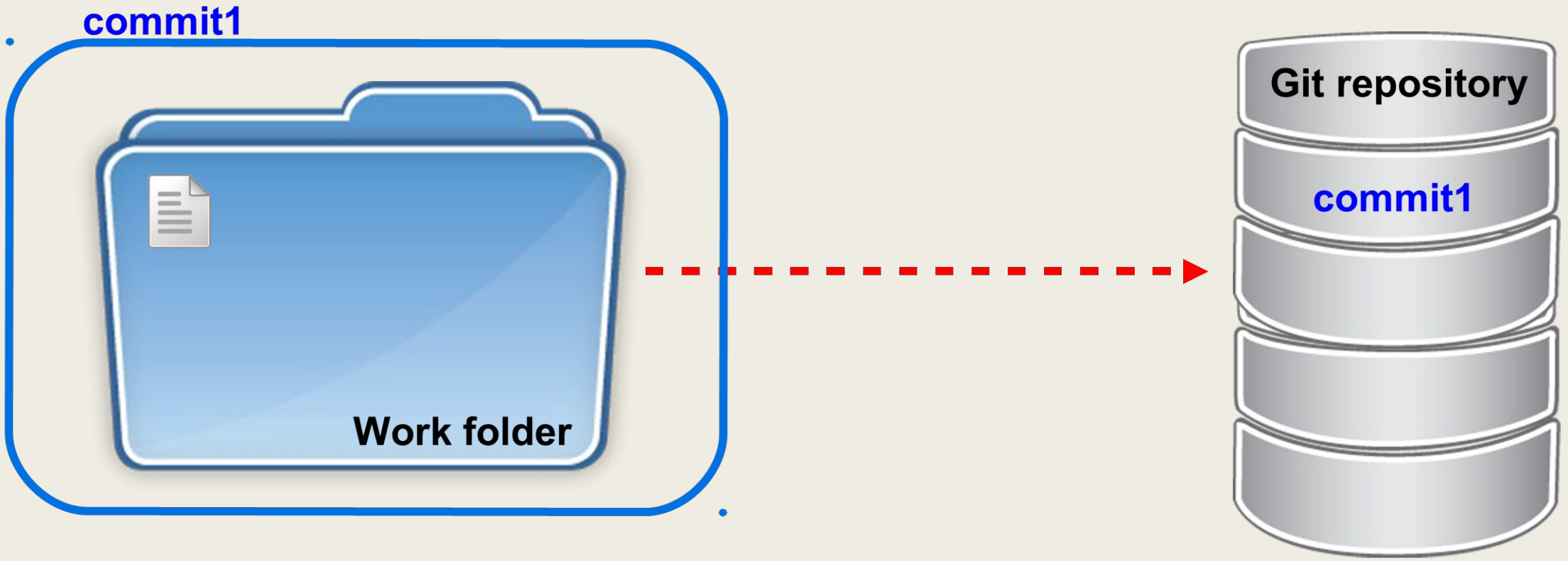
As we add/delete/edit files in our work folder, we can tell git to store a new ‘snapshot’ of our folders current state (a bit like saving a computer game)

Git allows us to make backups or ‘saves’ (called **‘commits’**) of our work that we can easily revert to if a problem occurs…



As we add/delete/edit files in our work folder, we can tell git to store a new ‘snapshot’ of our folders current state (a bit like saving a computer game)

Git allows us to make backups or ‘saves’ (called **‘commits’**) of our work that we can easily revert to if a problem occurs…



As we add/delete/edit files in our work folder, we can tell git to store a new ‘snapshot’ of our folders current state (a bit like saving a computer game)

Git allows us to make backups or ‘saves’ (called **‘commits’**) of our work that we can easily revert to if a problem occurs…

As we add/delete/edit files in our work folder, we can tell git to store a new ‘snapshot’ of our folders current state (a bit like saving a computer game)

Git allows us to make backups or ‘saves’ (called **‘commits’**) of our work that we can easily revert to if a problem occurs…

As we add/delete/edit files in our work folder, we can tell git to store a new ‘snapshot’ of our folders current state (a bit like saving a computer game)

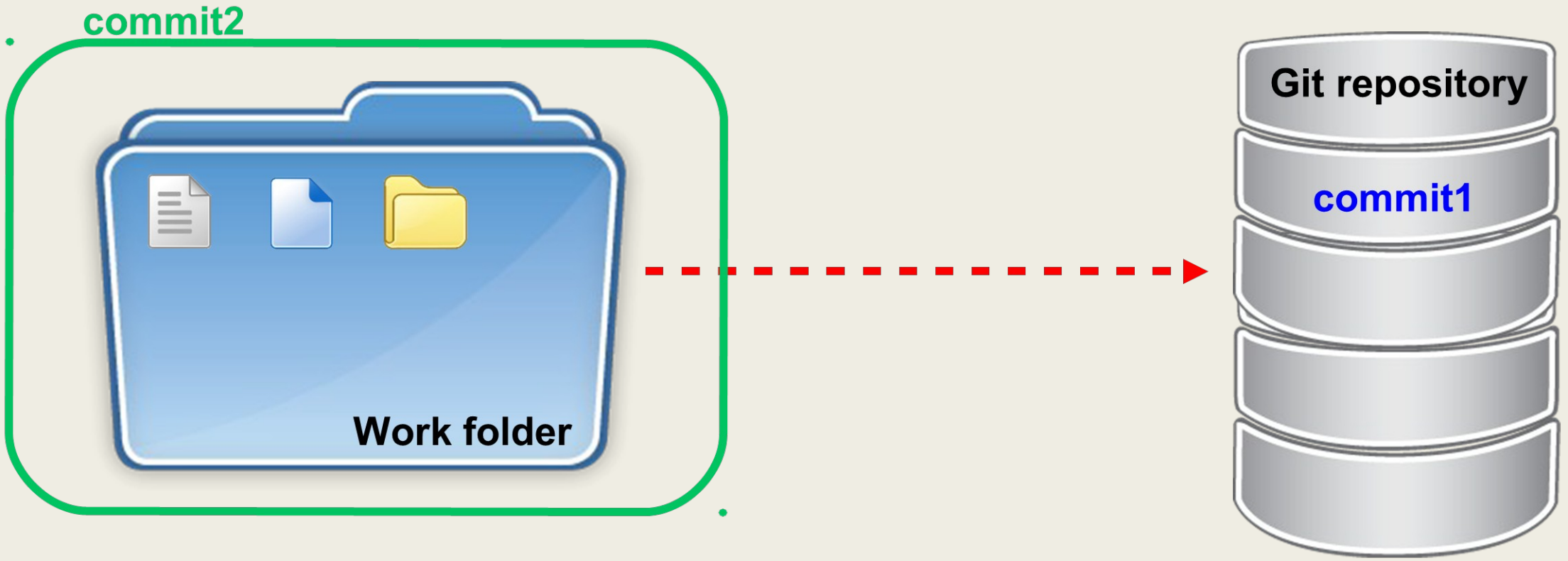
Git allows us to make backups or ‘saves’ (called **‘commits’**) of our work that we can easily revert to if a problem occurs…

As we add/delete/edit files in our work folder, we can tell git to store a new ‘snapshot’ of our folders current state (a bit like saving a computer game)

Git allows us to make backups or ‘saves’ (called **‘commits’**) of our work that we can easily revert to if a problem occurs…

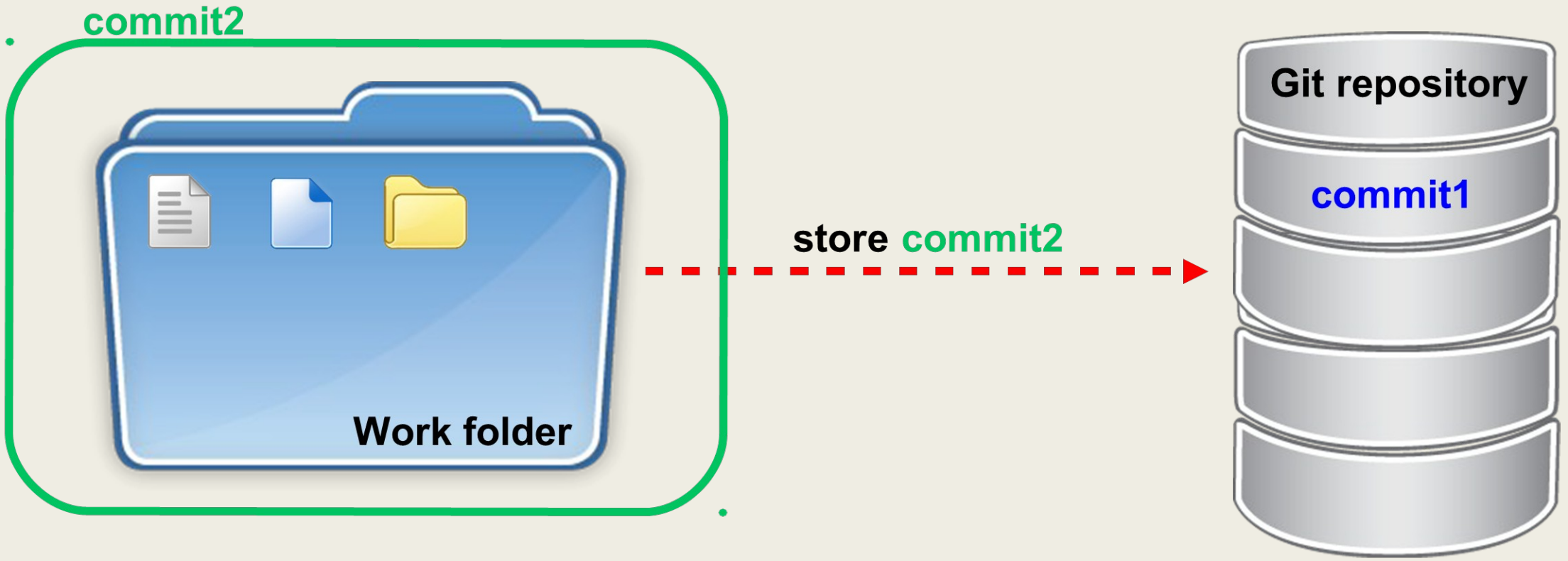
As we add/delete/edit files in our work folder, we can tell git to store a new ‘snapshot’ of our folders current state (a bit like saving a computer game)

Git allows us to make backups or ‘saves’ (called **‘commits’**) of our work that we can easily revert to if a problem occurs…



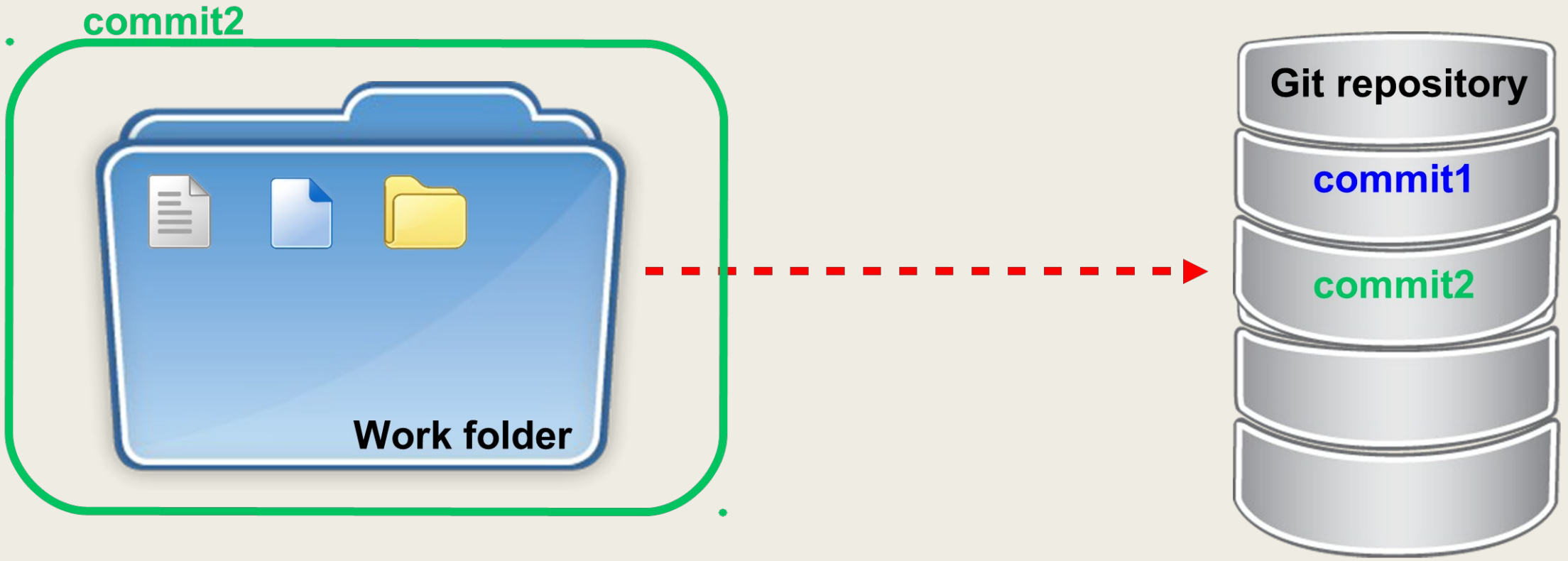
As we add/delete/edit files in our work folder, we can tell git to store a new ‘snapshot’ of our folders current state (a bit like saving a computer game)

Git allows us to make backups or ‘saves’ (called **‘commits’**) of our work that we can easily revert to if a problem occurs…



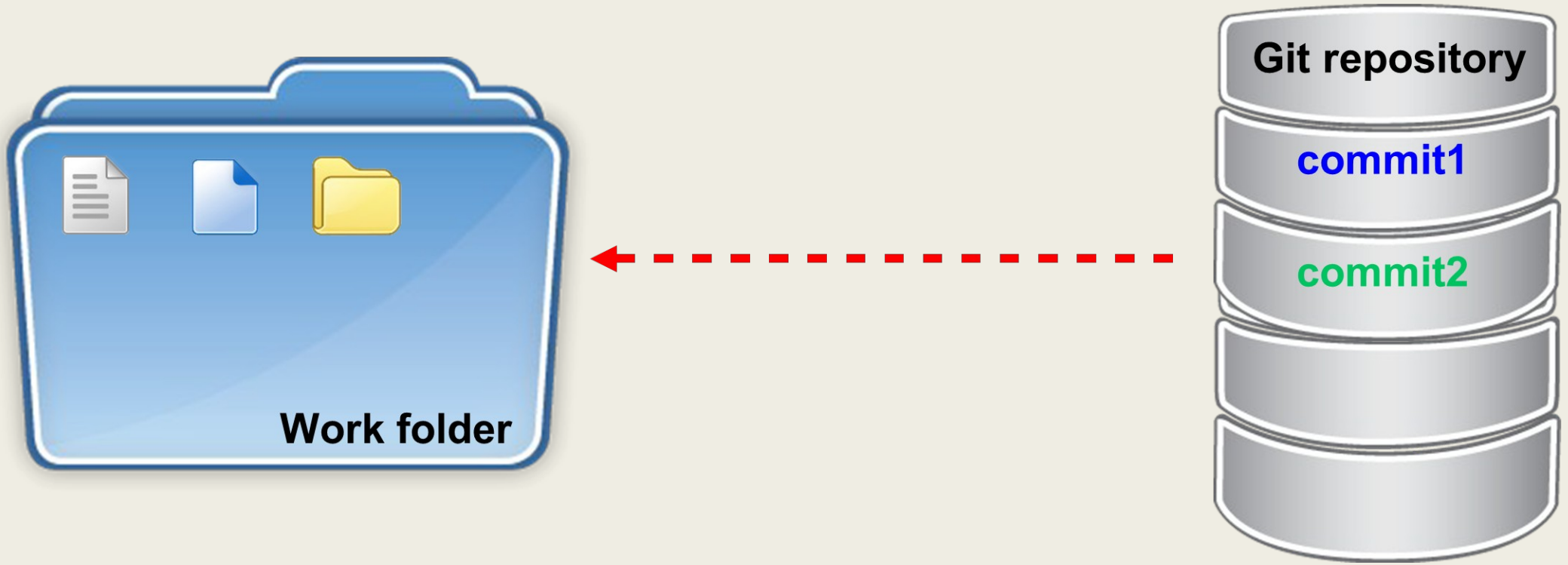
As we add/delete/edit files in our work folder, we can tell git to store a new ‘snapshot’ of our folders current state (a bit like saving a computer game)

Git allows us to make backups or ‘saves’ (called **‘commits’**) of our work that we can easily revert to if a problem occurs…

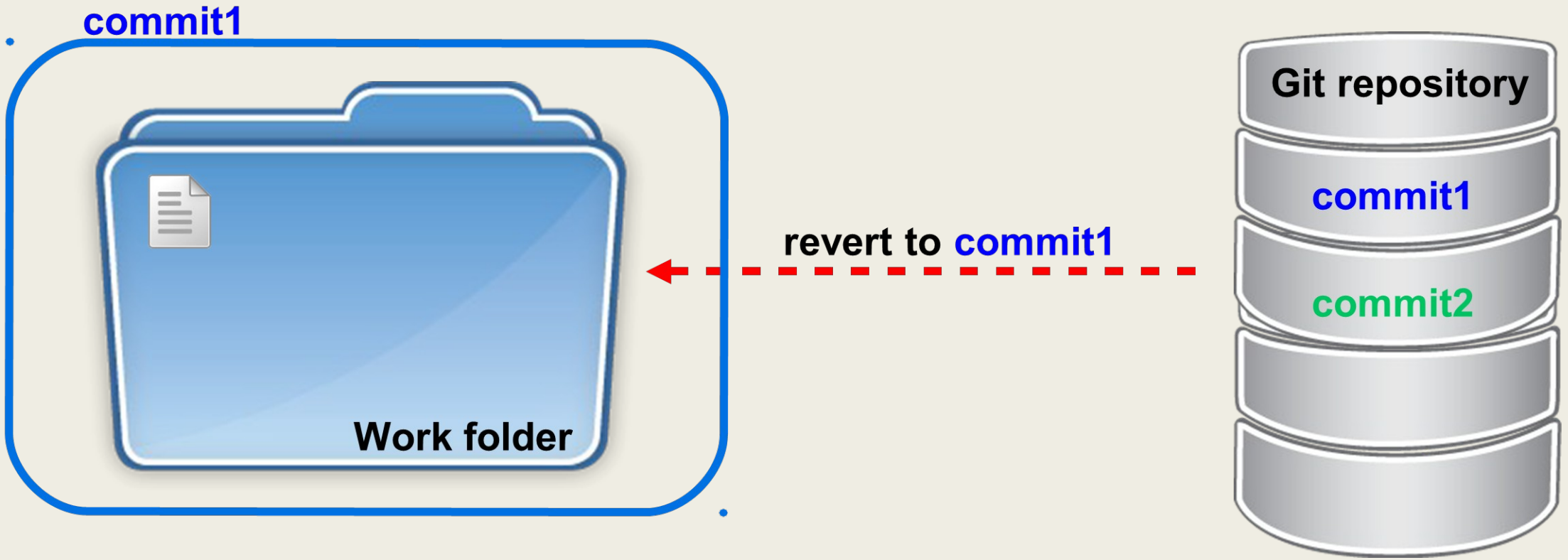


As we add/delete/edit files in our work folder, we can tell git to store a new ‘snapshot’ of our folders current state (a bit like saving a computer game)

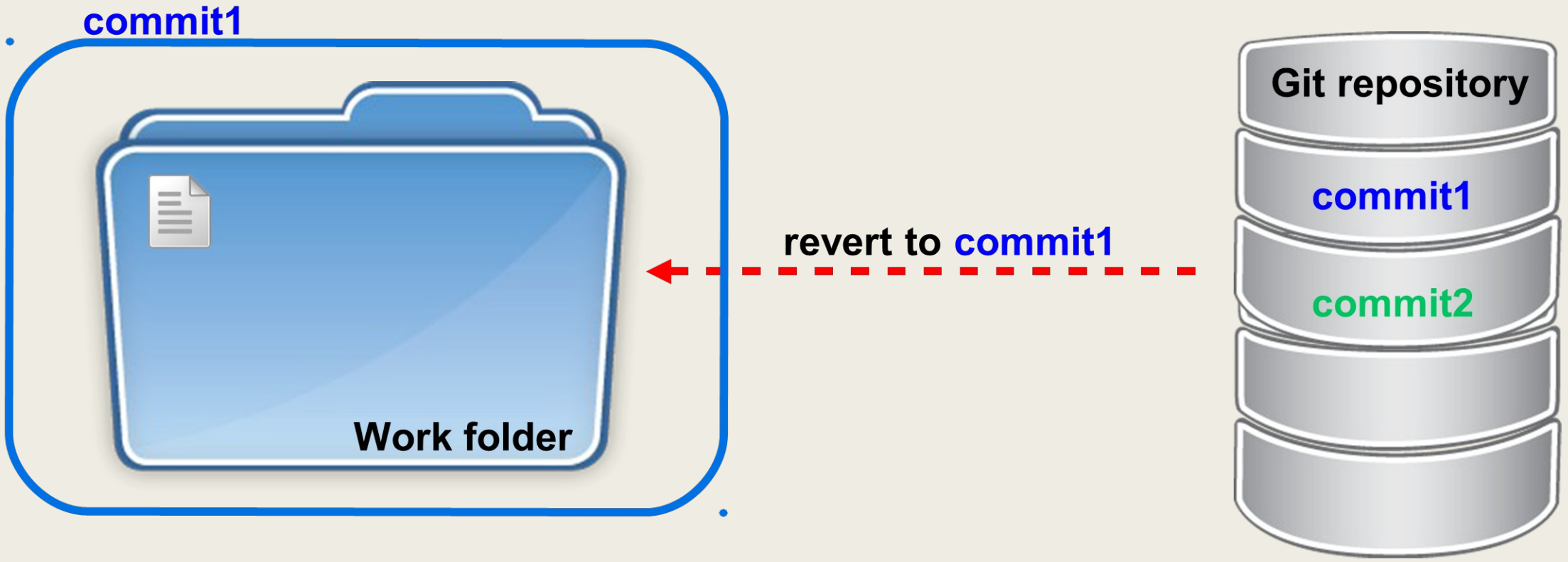
In the other direction, if we decide we need to drop the changes made in **commit2** and go back to **commit1**, we can:



In the other direction, if we decide we need to drop the changes made in **commit2** and go back to **commit1**, we can:



In the other direction, if we decide we need to drop the changes made in **commit2** and go back to **commit1**, we can:



Note that **commit2** is not lost. It will remain in the repository, and we can revert back to it if necessary.

This sort of functionality would have made things *much much easier* for the student...

This sort of functionality would have made things *much much easier* for the student...

So how do we get

- To create a repository that tracks changes in an existing folder structure named ‘A’, we write:

```
jakob@jakob-T430:~$ git init A
Initialized empty Git repository in /home/jakob/A/.git/
```

- To create a repository that tracks changes in an existing folder structure named ‘A’, we write:

```
jakob@jakob-T430:~$ git init A
Initialized empty Git repository in /home/jakob/A/.git/
```

- This turns folder A into a **working directory (WD)** and creates a **repository** in a hidden folder, named ‘*.git*’

```
jakob@jakob-T430:~/A$ ls
fileA.txt  fileB.txt  fileC.txt  fileD.txt  folder1
jakob@jakob-T430:~/A$ ls -a
.   fileA.txt  fileC.txt  folder1
..  fileB.txt  fileD.txt  .git
```

- To create a repository that tracks changes in an existing folder structure named ‘A’, we write:

```
jakob@jakob-T430:~$ git init A
Initialized empty Git repository in /home/jakob/A/.git/
```

- This turns folder A into a **working directory (WD)** and creates a **repository** in a hidden folder, named ***‘.git’***

```
jakob@jakob-T430:~/A$ ls
fileA.txt  fileB.txt  fileC.txt  fileD.txt  folder1
jakob@jakob-T430:~/A$ ls -a
.   fileA.txt  fileC.txt  folder1
..  fileB.txt  fileD.txt  .git
```

– *A working directory is a folder structure containing the **files you’ll be working on**. Can be thought of as a ‘sandbox’.*

- To create a repository that tracks changes in an existing folder structure named ‘A’, we write:

```
jakob@jakob-T430:~$ git init A
Initialized empty Git repository in /home/jakob/A/.git/
```

- This turns folder A into a **working directory (WD)** and creates a **repository** in a hidden folder, named ***‘.git’***:

```
jakob@jakob-T430:~/A$ ls
fileA.txt  fileB.txt  fileC.txt  fileD.txt  folder1
jakob@jakob-T430:~/A$ ls -a
.   fileA.txt  fileC.txt  folder1
..  fileB.txt  fileD.txt  .git
```

- *A working directory is a folder structure containing the **files you’llbe working on**. Can be thought of as a ‘sandbox’.*

- *The repository for a given WD is the .git folder stored inside that WD; this is where all the **information about previous file versions is stored**. It is a hidden folder (that’s why we use ‘**`ls -a`**’ to see it)*

- Imagine that we've modified files A-C, but not file D:

```
jakob@jakob-T430:~/A$ ls
fileA.txt  fileB.txt  fileC.txt  fileD.txt  folder1
jakob@jakob-T430:~/A$ ls -a
.   fileA.txt  fileC.txt  folder1
..  fileB.txt  fileD.txt  .git
```

- We can verify this with the command '***`git status`***':

```
jakob@jakob-T430:~/A$ 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:   fileA.txt
        modified:   fileB.txt
        modified:   fileC.txt

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

More terminology

- If we want to save the modifications we've made to files A-C, we **stage changes to** those files:

More terminology

- If we want to save the modifications we've made to files A-C, we **stage changes to** those files:
  - ***Staging*** *is how we tell Git which changes we want to include in our next save (commit); when we* ***stage a file*** *it goes to the* ***staging area***

More terminology

- If we want to save the modifications we've made to files A-C, we **stage changes to** those files:

  - ***Staging*** *is how we tell Git which changes we want to include in our next save (commit); when we* ***stage a file*** *it goes to the* ***staging area***

The staging area can be seen as a preview of the next commit

- We write '*git add <insert file names>*' to add files to the staging area

```
jakob@jakob-T430:~/A$ git add fileA.txt fileB.txt fileC.txt
jakob@jakob-T430:~/A$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   fileA.txt
        modified:   fileB.txt
        modified:   fileC.txt
```

- We write '*git add <insert file names>*' to add files to the staging area

```
jakob@jakob-T430:~/A$ git add fileA.txt fileB.txt fileC.txt
jakob@jakob-T430:~/A$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   fileA.txt
        modified:   fileB.txt
        modified:   fileC.txt
```

- This tells git that all changes to files A-C are to be included in the next commit.

- We write '*git add <insert file names>*' to add files to the staging area

```
jakob@jakob-T430:~/A$ git add fileA.txt fileB.txt fileC.txt
jakob@jakob-T430:~/A$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   fileA.txt
        modified:   fileB.txt
        modified:   fileC.txt
```

- This tells git that all changes to files A-C are to be included in the next commit.
- The command 'git status' shows us which files are in the staging area and which are not.

- We write '*git add <insert file names>*' to add files to the staging area

```
jakob@jakob-T430:~/A$ git add fileA.txt fileB.txt fileC.txt
jakob@jakob-T430:~/A$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   fileA.txt
        modified:   fileB.txt
        modified:   fileC.txt
```

We can also **unstage** files (i.e., remove from the staging area) using the '***git reset ...***' command

- This tells git that all changes to files A-C are to be included in the next commit.
- The command 'git status' shows us which files are in the staging area and which are not.

How to make a commit

- A commit is a ‘**snapshot**’ of the current state of your working directory

How to make a commit

- A commit is a '**snapshot**' of the current state of your working directory

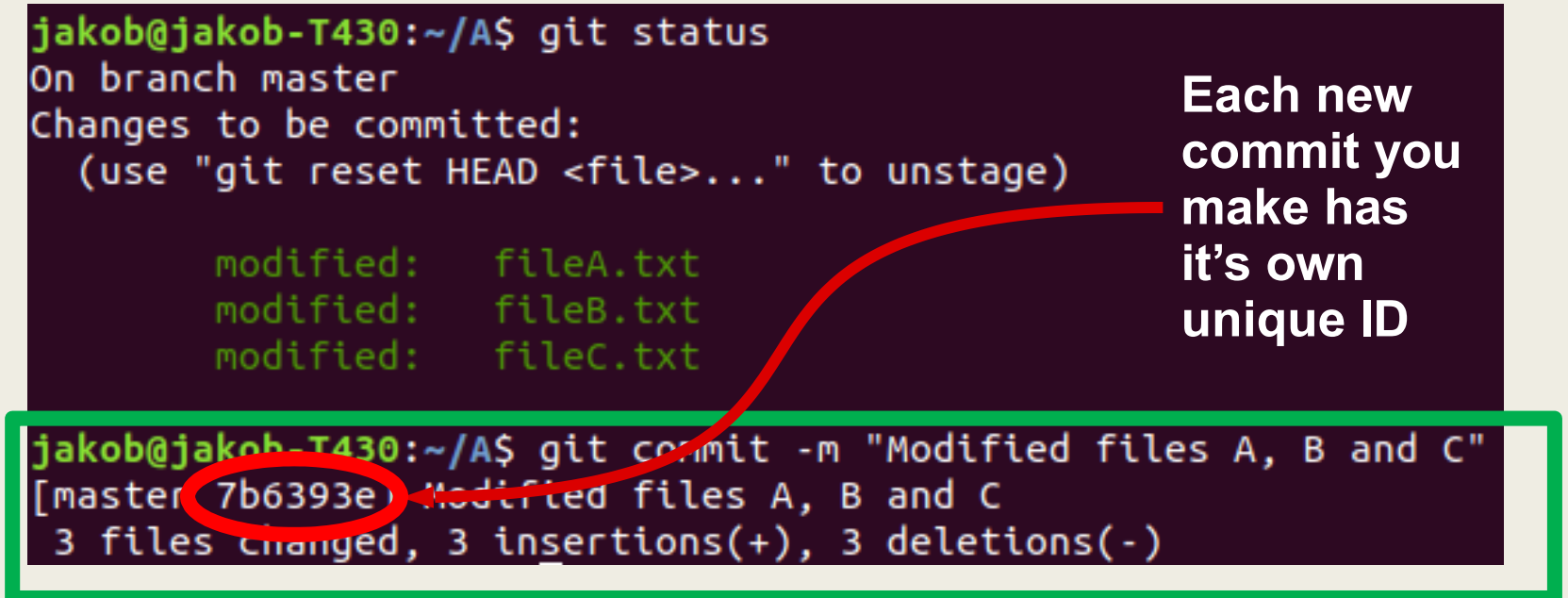
- 'git commit ...', adds (to the repository) a snapshot with all currently staged changes and all unchanged content from the previous commit

```
jakob@jakob-T430:~/A$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   fileA.txt
        modified:   fileB.txt
        modified:   fileC.txt

jakob@jakob-T430:~/A$ git commit -m "Modified files A, B and C"
[master 7b6393e] Modified files A, B and C
 3 files changed, 3 insertions(+), 3 deletions(-)
```

How to make a commit

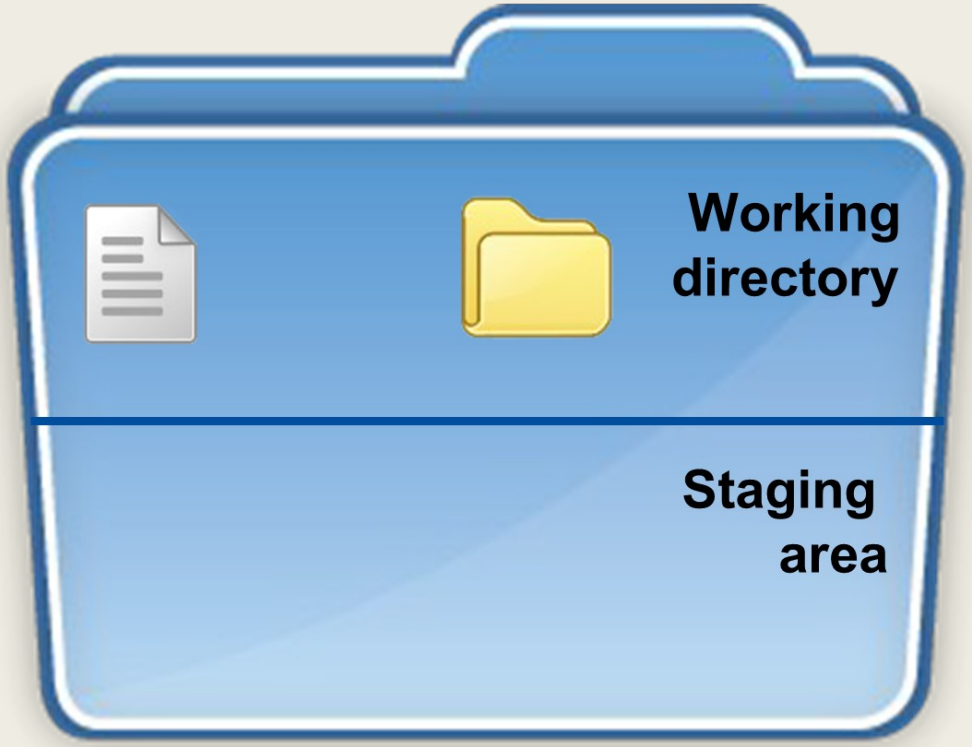
- A commit is a '**snapshot**' of the current state of your working directory

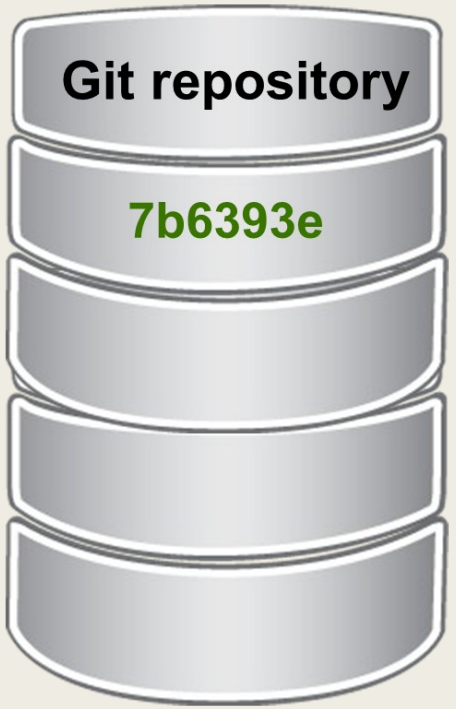
- 'git commit ...', adds (to the repository) a snapshot with all currently staged changes and all unchanged content from the previous commit



Staging and committing changes

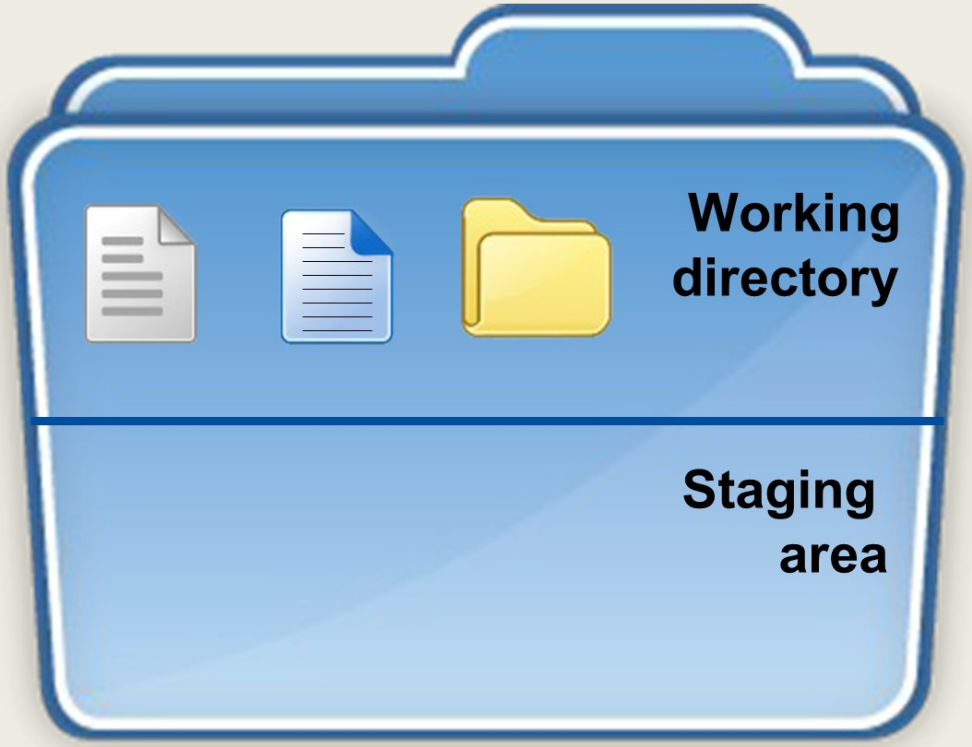
Once we reach a ‘milestone’ in our WD, we stage changes using `‘git add . . .’`
We then make a commit containing both modified/unmodified files

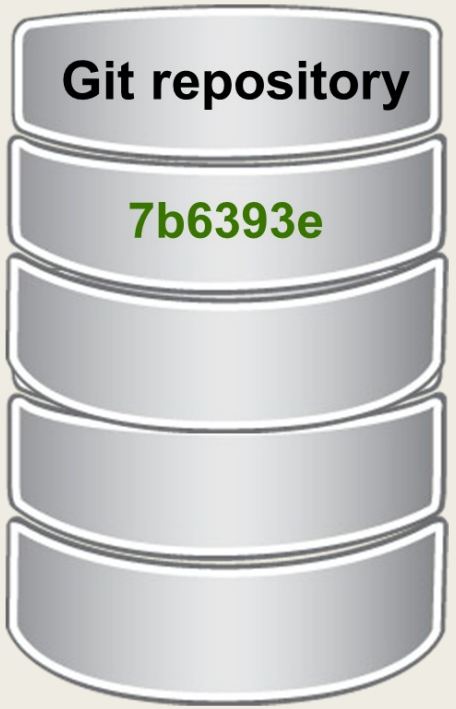




Staging and committing changes

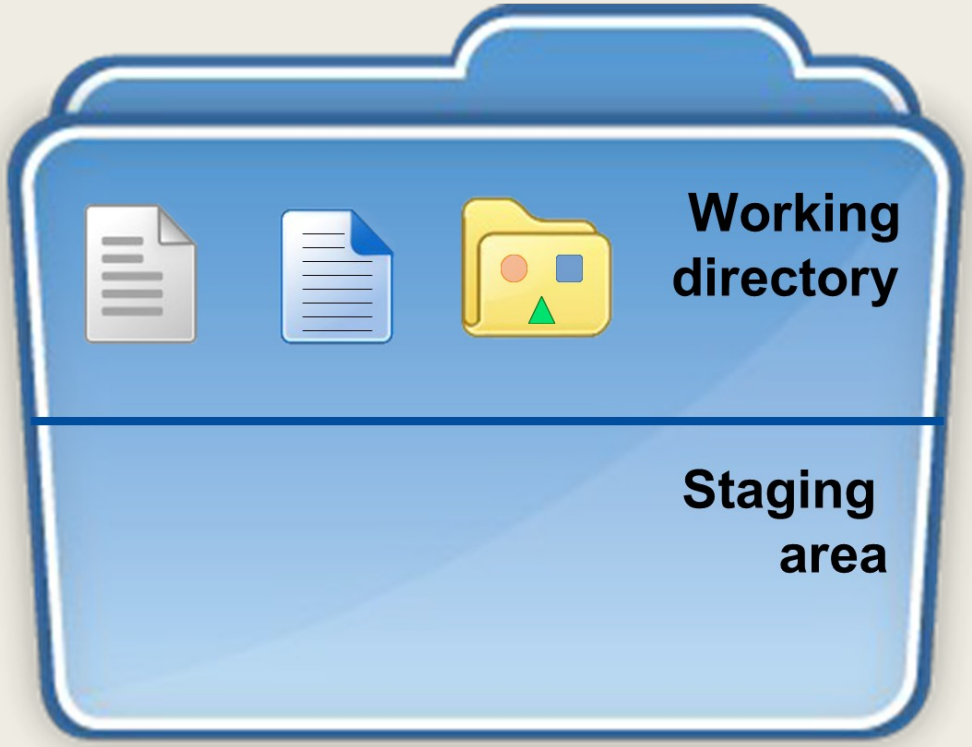
Once we reach a ‘milestone’ in our WD, we stage changes using `‘git add . . .’`
We then make a commit containing both modified/unmodified files

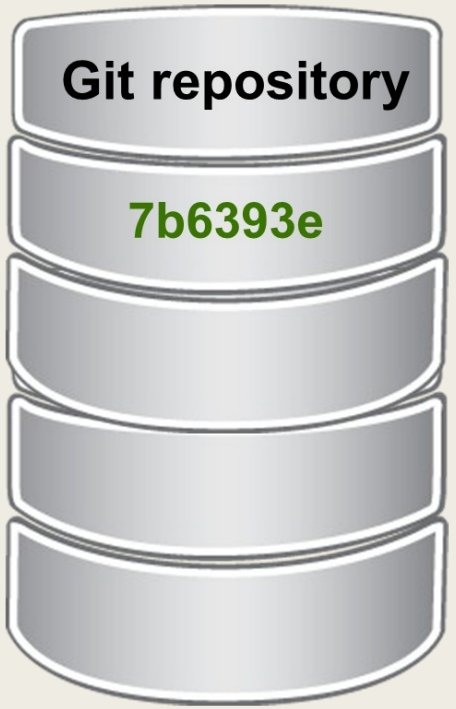




Staging and committing changes

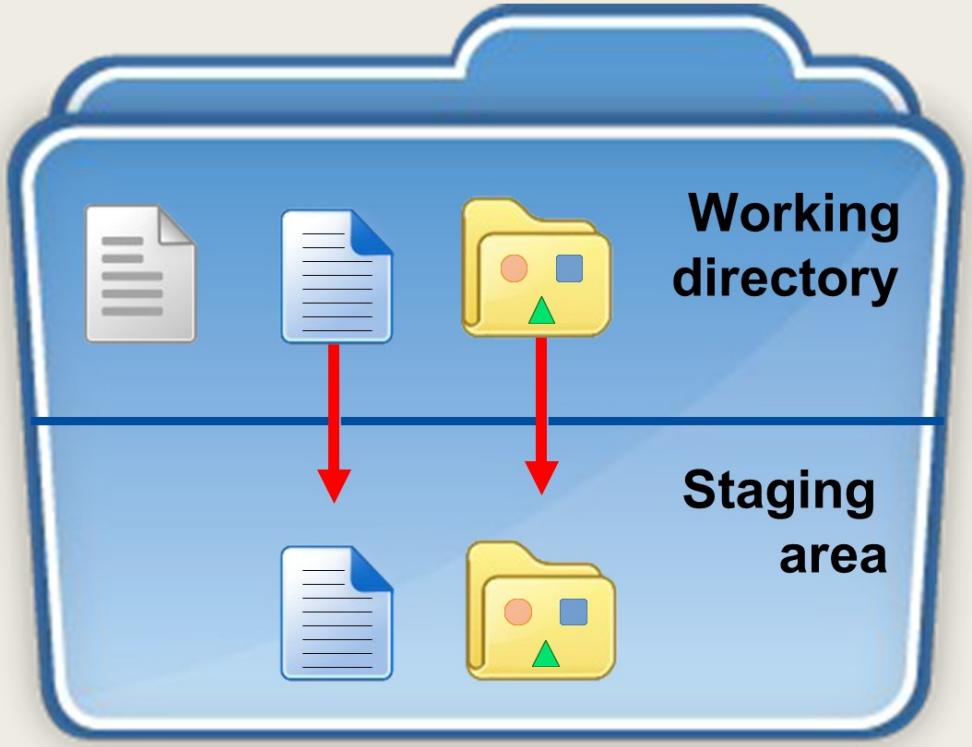
Once we reach a ‘milestone’ in our WD, we stage changes using `‘git add . . .’`
We then make a commit containing both modified/unmodified files

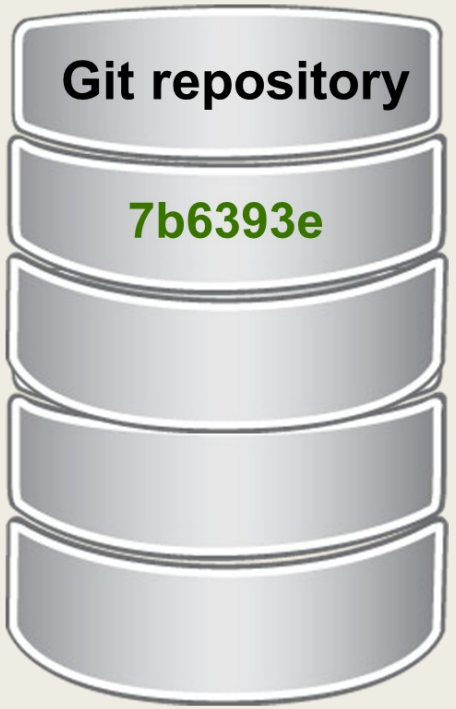




Staging and committing changes

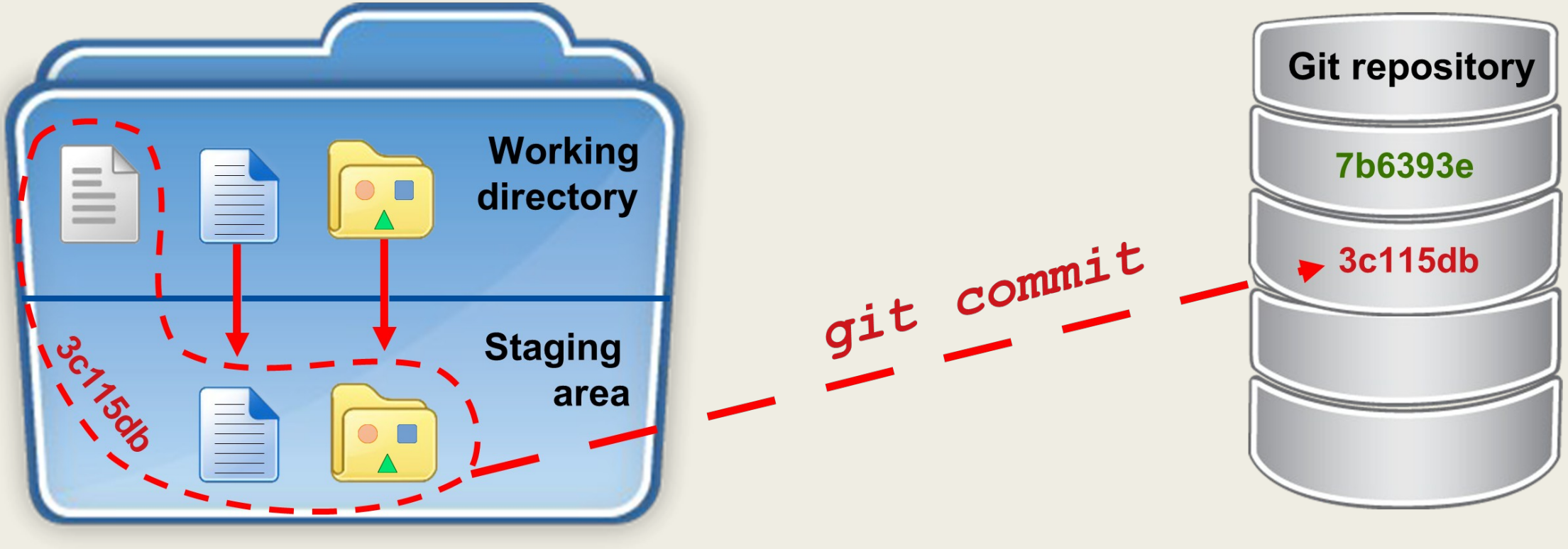
Once we reach a ‘milestone’ in our WD, we stage changes using ‘`git add . . .`’
We then make a commit containing both modified/unmodified files





Staging and committing changes

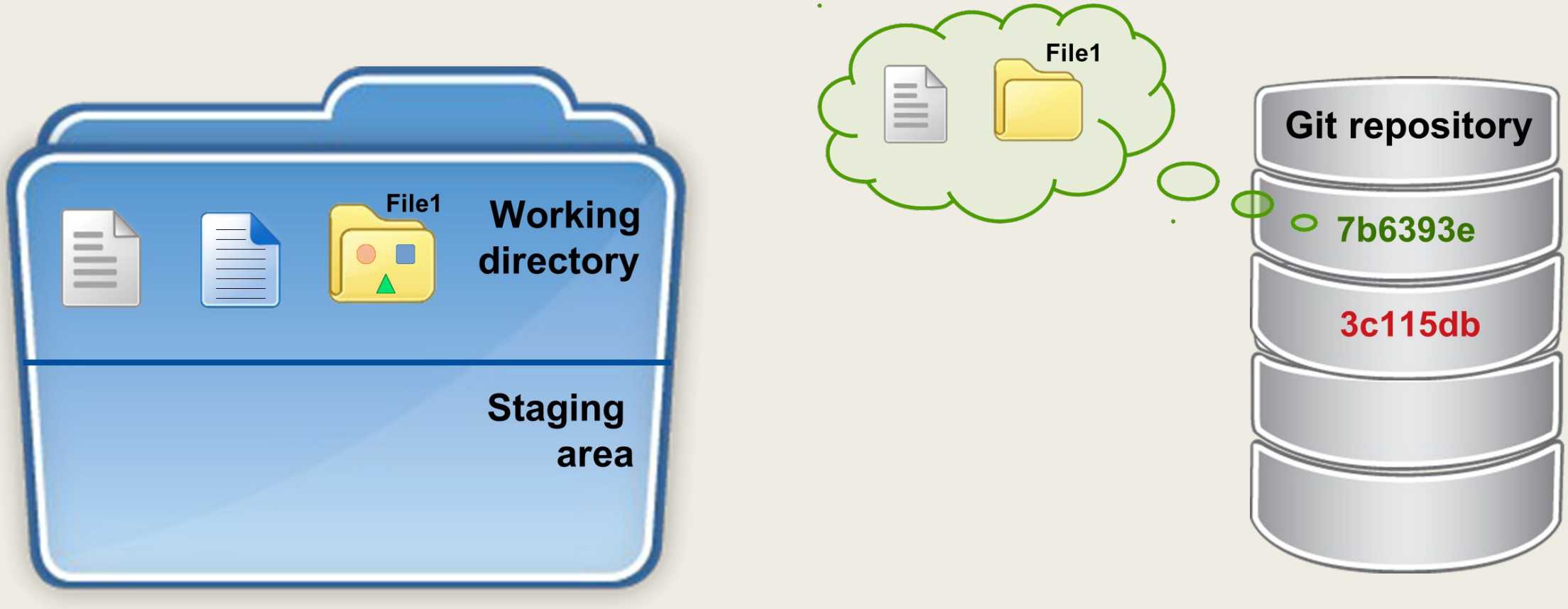
Once we reach a ‘milestone’ in our WD, we stage changes using ‘`git add . . .`’
We then make a commit containing both modified/unmodified files



Versions of unstaged files from the previous commit are **also included in the new commit**

Retrieving an old file version

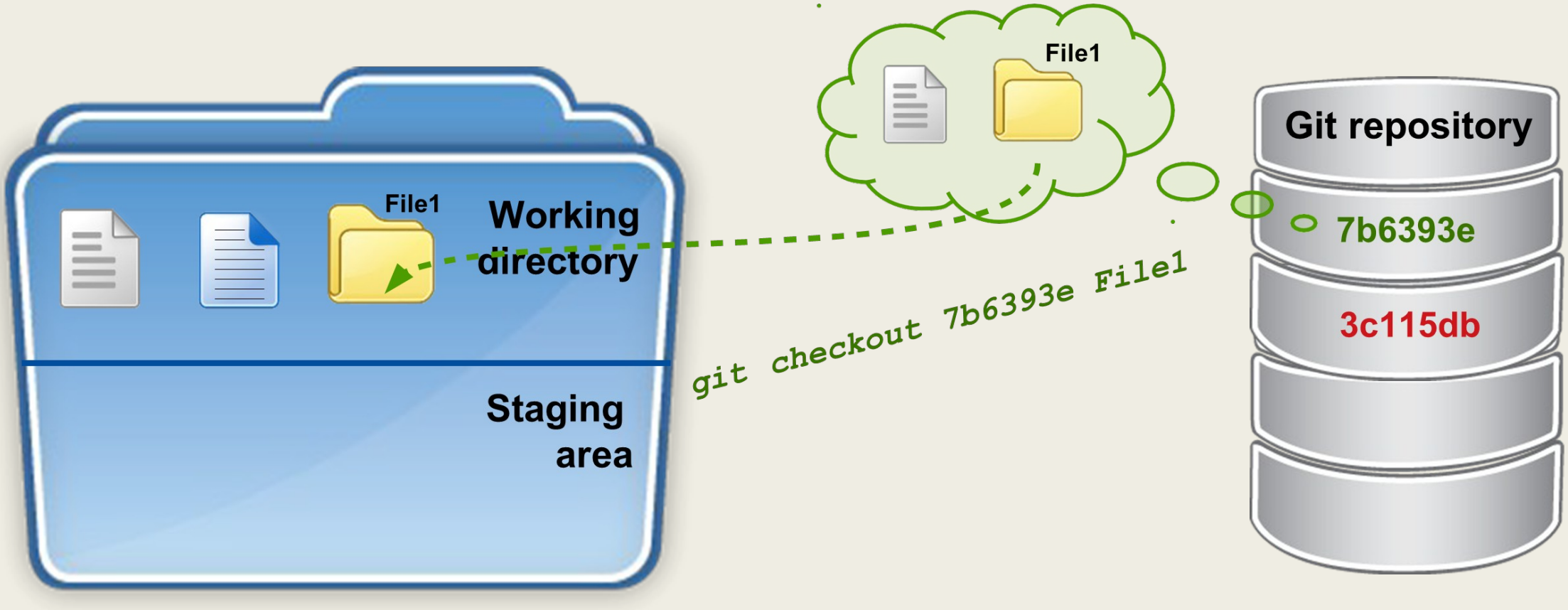
If we now wish to revert File1 back to the version stored in commit 7b6393e, we can use the command: '`git checkout <ID_of_commit> <name_of_file>`'



The retrieved old version of the file is added to the staging area, and we make a new commit

Retrieving an old file version

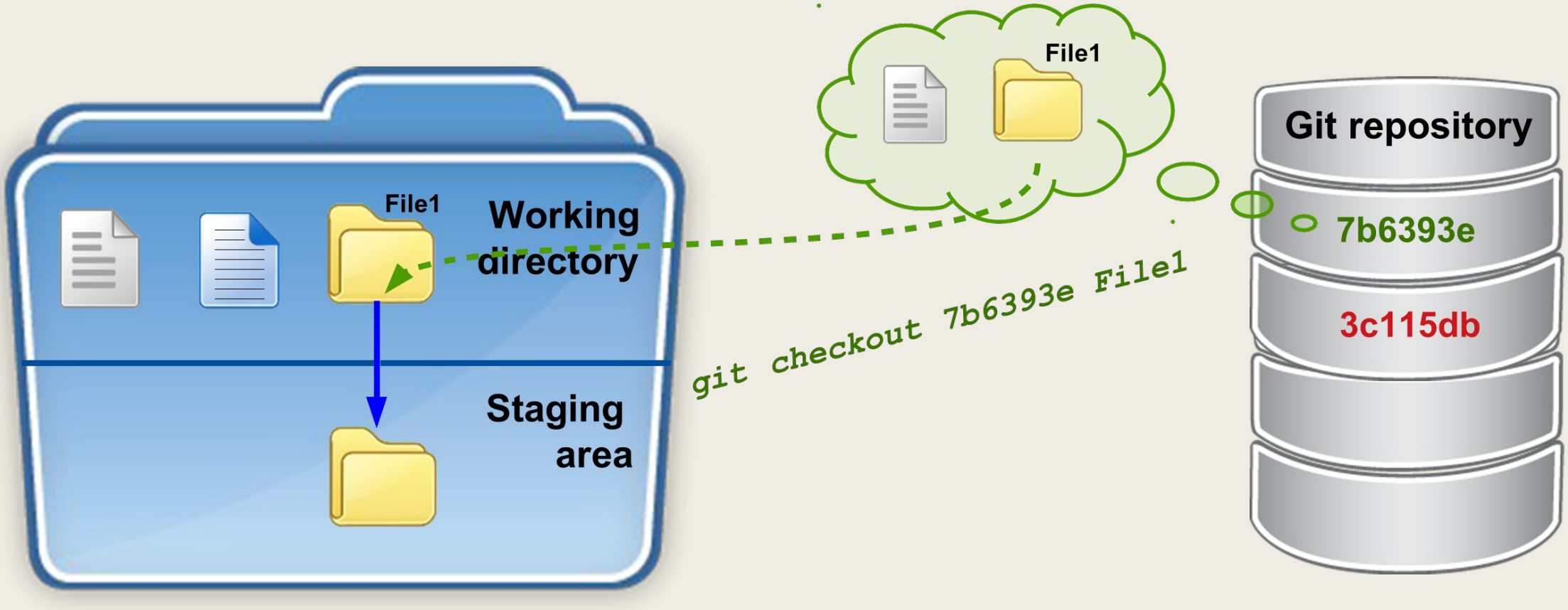
If we now wish to revert File1 back to the version stored in commit 7b6393e, we can use the command: '**`git checkout <ID_of_commit> <name_of_file>`**'



The retrieved old version of the file is added to the staging area, and we make a new commit

Retrieving an old file version

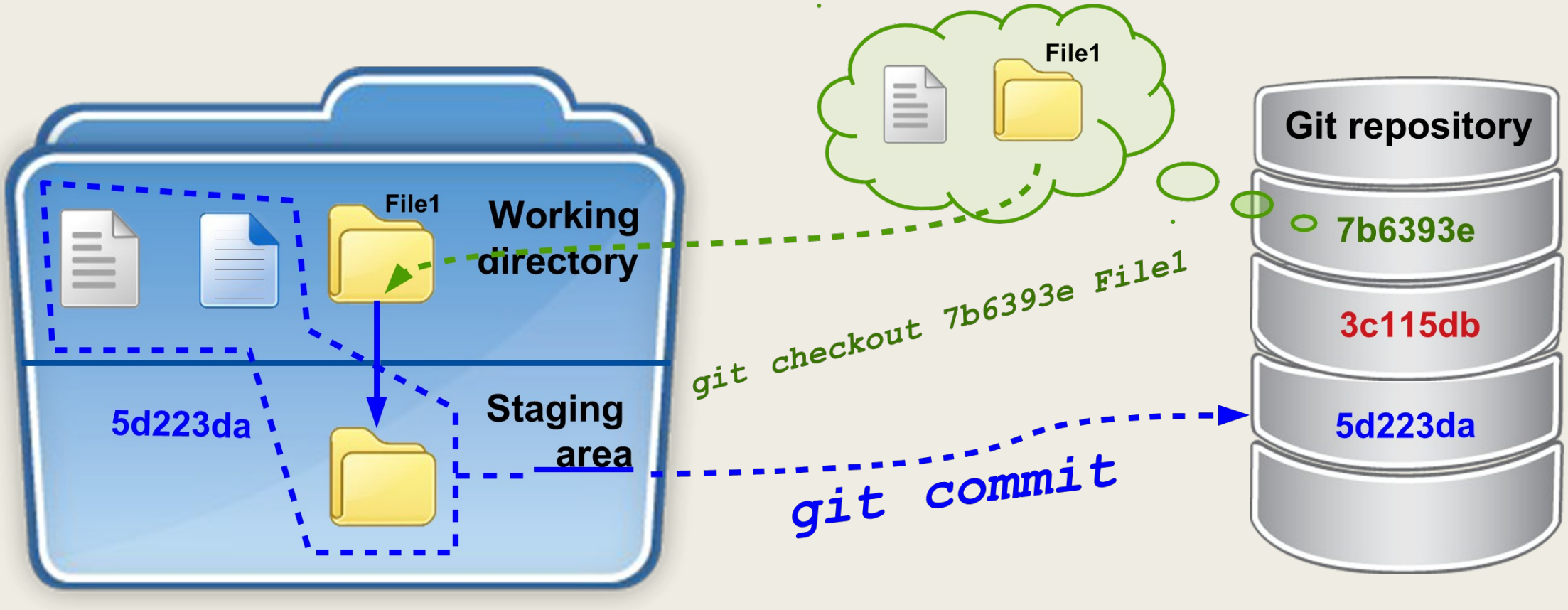
If we now wish to revert File1 back to the version stored in commit 7b6393e, we can use the command: '**`git checkout <ID_of_commit> <name_of_file>`**'



The retrieved old version of the file is added to the staging area, and we make a new commit

Retrieving an old file version

If we now wish to revert File1 back to the version stored in commit 7b6393e, we can use the command: '**`git checkout <ID_of_commit> <name_of_file>`**'



The retrieved old version of the file is added to the staging area, and we make a new commit

Retrieving an old file version

- To retrieve an old version of a file we need the unique ID of the commit that contains that version. To find that out we use '`git log` ...'

```
jakob@jakob-T430:~/A$ git log --oneline
7b6393e (HEAD -> master) Modified files A, B and C
23e7dab Modified files A, B, C
455f978 Modified fileB.txt
e36c65b Modified fileB.txt
ca3add3 changes
297b095 Fixed bug in fileB.txt
```

- The command provides a list of all commits in the repository history, and we can select the one containing the version of the file we wish to revert to:

```
jakob@jakob-T430:~/A$ git checkout 7b6393e fileA.txt
jakob@jakob-T430:~/A$ git commit -m "reverted fileA.txt to version 7b6393e"
[master cfcd484] reverted fileA.txt to version 7b6393e
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Up to this point, we know that git

- Allows us to back up work and easily revert to older copies of file without rewriting more recent change history
- Aggregates details of modifications to any file in folder structure into one view and offers a wide array of commands/functionality
- Is free and relatively easy to use with a little practice
- Sees widespread usage within the software development sector

Up to this point, we know that git

- Allows us to back up work and easily revert to older copies of file without rewriting more recent change history
- Aggregates details of modifications to any file in folder structure into one view and offers a wide array of commands/functionality
- Is free and relatively easy to use with a little practice
- Sees widespread usage within the software development sector

What's next? File sharing and collaboration in git

Git as a collaboration tool

- So far we looked at how git can be useful **locally** for file history-tracking and backup
- Arguably, Git's main usage comes from developers looking to collaborate on files across the internet
- A '**remote repo**' allows us to store project files in a single centralised place where each developer on the project can access them

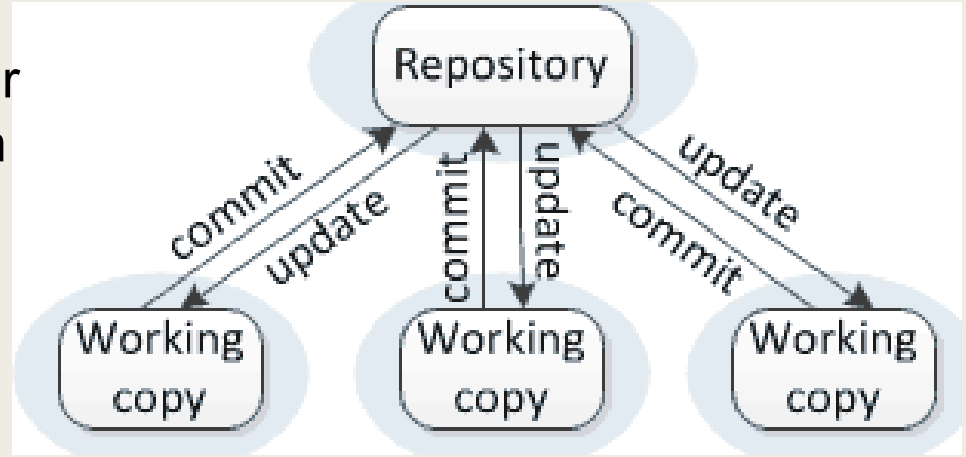
What is a remote repo?

- Also called a **remote**, It's a repository that's hosted on the internet
- We connect to a remote, make an isolated local copy of it, then make changes to the repo on our computer
- We can then 'sync' our local copy with the remote repo, **pushing** changes to the remote repo, and **pulling** changes from the remote repo

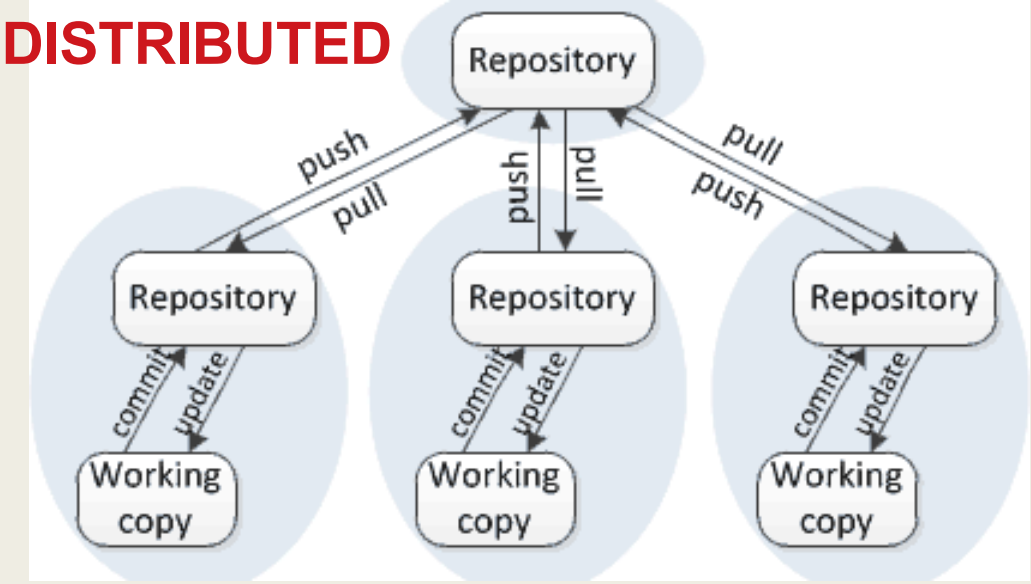
There are two main types of VCS...

CENTRALISED

Centralised VCS's maintain one central repository which is stored on a remote server that each collaborator directly interfaces with



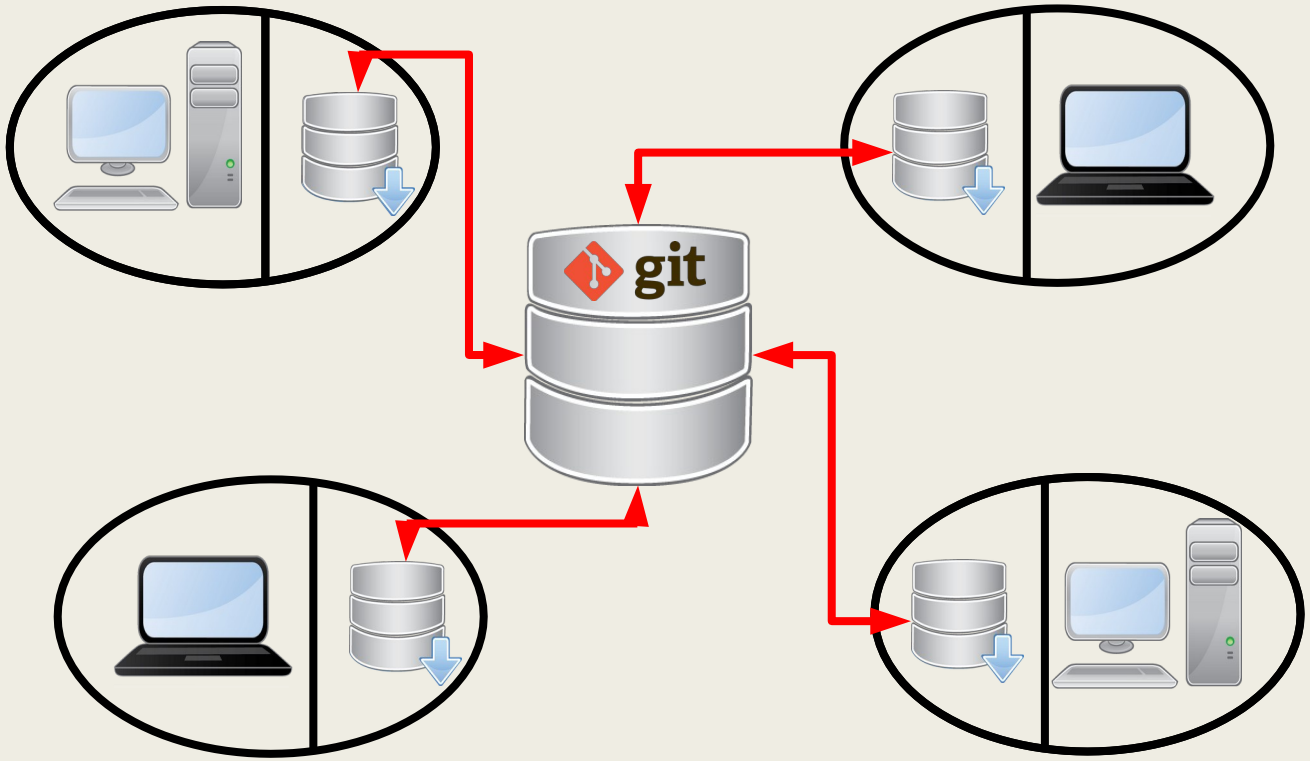
DISTRIBUTED



Distributed VCS's have a central remote repo, but each developer maintains their own isolated local copy

Communicating with a remote repo in git

Each person with access rights connects to the repo and stores a **clone** of it on their system, obtained by running a certain command.



They make local changes on their system, then **push** these changes to the remote repository

Once these changes have been pushed by one collaborator, other collaborators can **pull** them to their own local repository

Creating your own remote repo

- There are a number of options for creating remote repos; one of the most popular is '**GitHub**'
- We sign up to GitHub (or another service) create a repository on their hosted server, then modify the files either in the browser-based GUI, or by **cloning** a local copy to our desktop
- We can control who is able to access the remote repo and who can modify it

Let's see how to create a remote repository in 'GitHub'

How to clone a remote repo

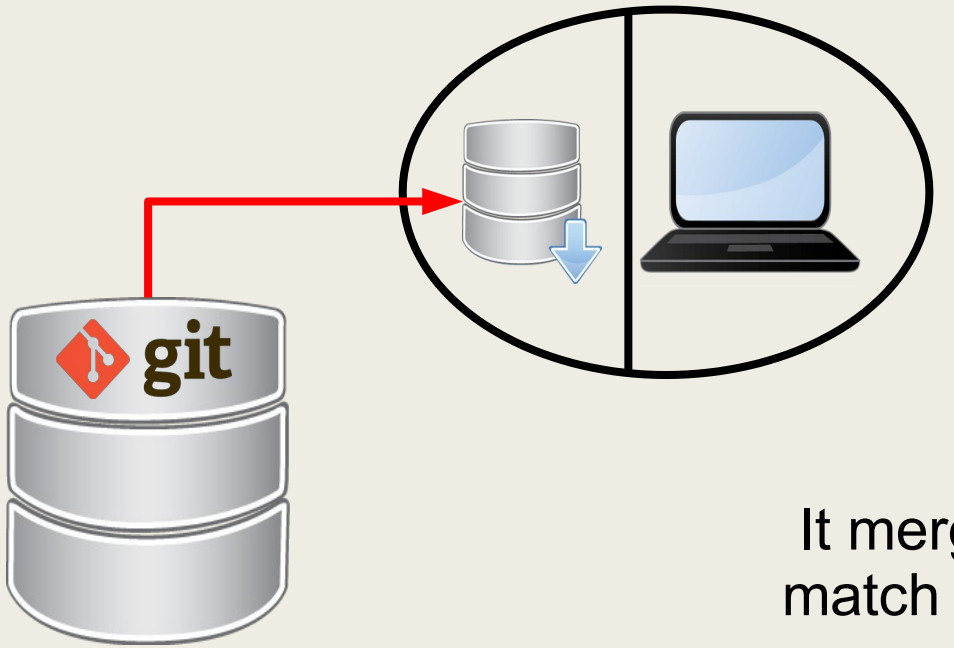
- Once we've created a remote repo, we can then use the command
- '**`git clone <url_of_remote>`**'
- to create a local copy of the repository and update your working directory:

```
jakob@jakob-IdeaPad-5-15ITL05:~$ git clone https://github.com/jtspooner93/test
Cloning into 'test'...
Username for 'https://github.com': jtspooner93
Password for 'https://jtspooner93@github.com':
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), 1.24 KiB | 1.24 MiB/s, done.
```

- We can then work on the cloned repository using the same commands that we've already seen (i.e., git commit, git checkout, git add etc...)

‘Pulling’ changes from a remote repo

If we wish to **update our local copy of a remote repository** with any changes/commits that others have made, we can use the ‘**`git pull`**’ command



‘**`git pull`**’ downloads all commits that are currently in the remote repo but not in your local repo

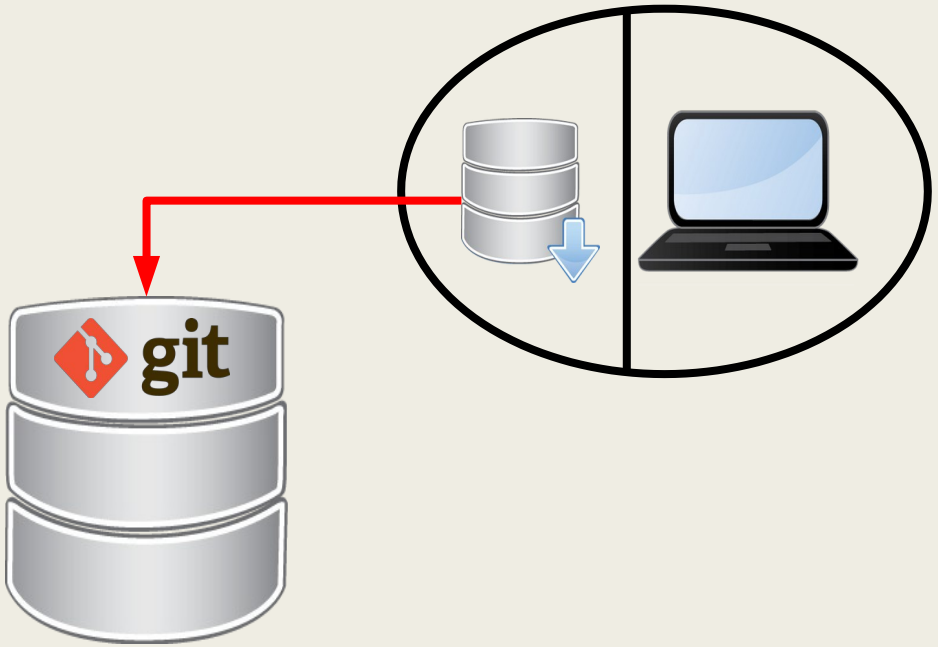
It merges the contents of your WD to match the contents of the latest commit

‘Pushing’ changes to a remote repo

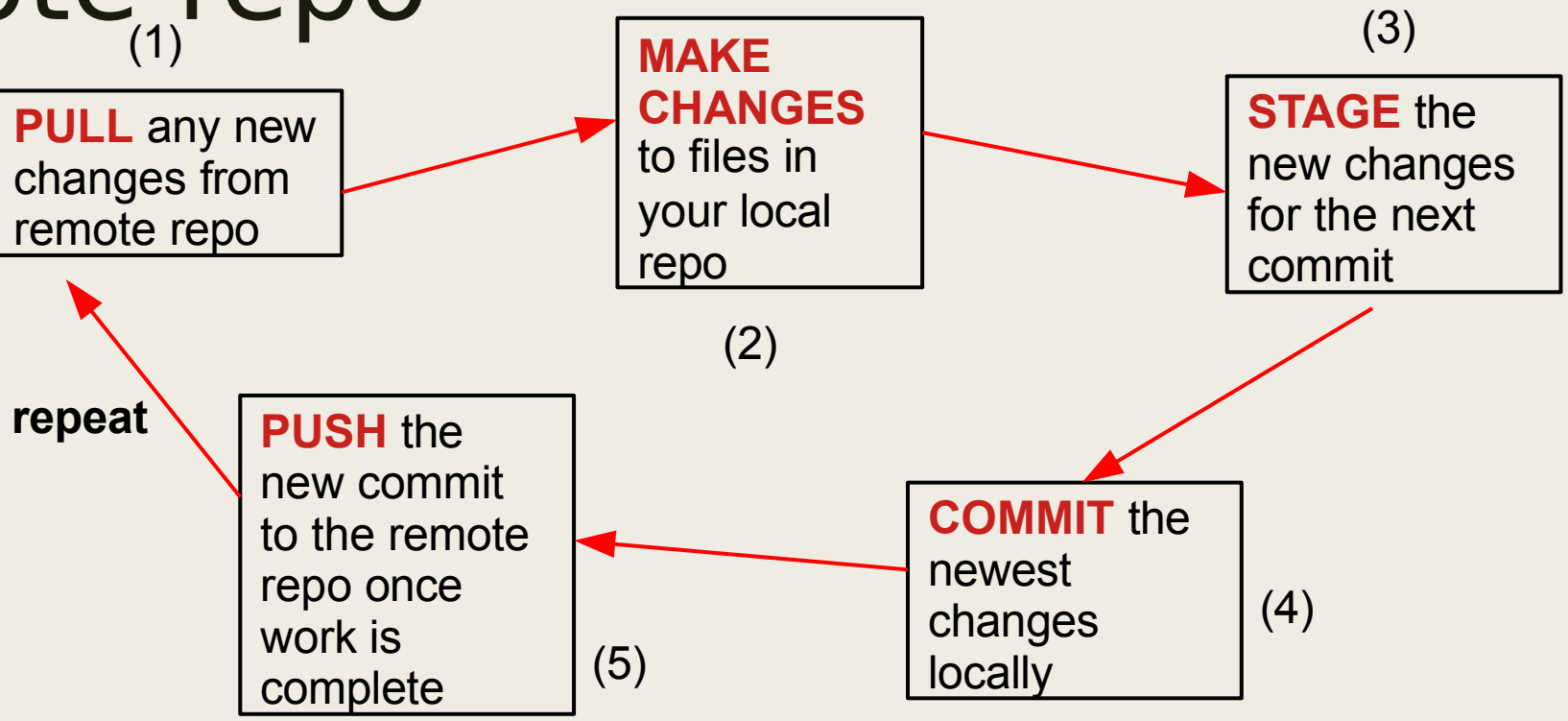
If we wish to **update the remote repo** with any changes/commits made to our local copy, we can use the ‘**`git push`**’ command

‘**`git push`**’ uploads all commits that are not contained in the remote repo but are contained in your local version

It’s good practice to always do ‘**`git pull`**’ and sync your repo with the changes of others before pushing!



Typical workflow in a cloned remote repo



It's important to try and coordinate your changes with teammates to avoid conflicts (we'll discuss this further next week)

What’s next?

- Follow a tutorial to check out your group repository and use git for submits and to resolve conflicts
- Tutorials this week in different rooms! (for PC access)
- Lecture next week:
  More advanced git topics

Questions?

- Dr. Matthias Heintz or Prof. Shigang Yue
  - *mmh21@leicester.ac.uk or sy237@Leicester.ac.uk*
  - *Microsoft Teams*
  - *Office 613 or 608 in Ken Edwards Building*

UNIVERSITY OF LEICESTER