# Теоретические основы GIT BUSH

## . Общая характеристика курса «Контроль версий с помощью Git»

Проблема синхронизации файлов:

1) Standalone work (Единоличная работа). Система хранения в разных директориях периодически обновляемого контента (наличие дополнений, ремарок и ревизий для одного исходного файла). ПРОБЛЕМА: сложность в навигации, определении авторства той или иной версии, даты создания каждой модификации и т.п.

2) Team work (Командная работа). В операционных системах предоставлена возможность “пошарить” искомую папку по сети (при наличии некоего сетевого диска). Можно одновременно править один файл, если разработчики имеют одинаковый сетевой location. ПРОБЛЕМА: Кто последний сохраняет редактируемый файл, тот перезаписывает чужие изменения, следовательно, часть новых данных внесённых или исправленных другими пользователями исчезнет.

3) Cloud (Облако) или Standalone/Team. Работа над файлом производится в облачном хранилище (storage) или хостинге (hosting). ПРОБЛЕМА: При одновременном внесении корректив в один файл двумя и более разработчиками, система не распознаёт какие изменения корректны, а какие — нет.

Данные проблемы призвана разрешить т.н. система версионного контроля (VCS). Ниже приведём цели данной системы:

1. Backup (создание резервной копии) и Restore (восстановление из копии)
2. Synchronization (организация совместной работы над некоторой проблемой (solution))
3. Undo (отмена (или откат) изменений и корректив)
4. Tracking changes (отслеживание изменений в реальном времени) и ownerships (выяснение авторства того или иного издания (edit) файла)
5. Sandboxing (возможность создания т.н. песочниц системы, где проводятся предварительные тесты и изменения, не повлияющие на структуру основного проекта).
6. Branching (Ветвление и слияние. Введение некоторой иерархии изменений)

По механизму работы с файлами системы версионного контроля разделяют на три группы (Объясним на примере с двумя девелоперами (Comp Alice и Bob)):

1. Lock-Modify-Unlock (Примеры: SCCS, Vault);

Парадигма блокировки доступа для всех пользователей, кроме одного (того, кто редактирует). Действия по праке происходят поочерёдно в порядке поступлени запроса.

1.1. Alice блокирует файл на условном репозитории хостинга (или в директории файловой системы); файл доступен остальным только для чтения;

1.2. Отправка корректив на сервер (commit);

1.3. Снятие блокировки (unlock);

1.4. Установка самостоятельной блокировки другим разработчиком (Bob);

1.5. Обновление локальной копии (в файловой системе. Перед правкой файла нужно подтянуть свежую версию с помощью команд update или pull);

1.6. Повтор шага 2;

1.7. Повтор шага 3.

1. Copy-Modify-Merge (Примеры: CVS, GIT);

Отсутствие возможности блокировки файла (все девелоперы одновременно правят файл без ограничений доступа). На сервере запускаются методы проверки на соответствие начальных версий на хостинге и на локальной директории пользователя.

2.1. Alice первая закончила своё издание (edit). commit (Alice);

2.2. На сервере изначально хранился пустой файл. Алгоритм: если начальная версия пользователя и сервера совпадают, то последний помещает редакцию Alice в свой файл. (Пусть Alice добавила 2 строчки рифмы (её исходный файл так же был пуст));

2.3. commit (Bob);

2.4. Начальный файл Bob (пустой) и только исправленный файл в сетевом репозитории (2 строчки песни Alice) не совпадают, изменения не проходят;

2.5. Сервер предлагает refresh (обновление) контента для Bob с текущего хостинга.

(при этом 2 строчки песни Bob остаются, а сам Bob производит pull (подтягивание) контента с сервера).

1. Combined (Примеры: SVN, TFS).

Помимо стратегий взаимодействия файлов сервера и девелопера, системы версионного контроля разделяют по особенностям работы с серверами (centralized и distributed).

***I. Centralized (централизованная)***   ***II. Distributed (распределённая)***

Central VCS-сервер на котором хранится VCS-сервер с историей версий.

история внесения поправок и редакций в У каждого пользователя есть своя

файл разработчиками (Version Database). база с историей и алгоритмами для

У девелоперов локально находятся по одному её просмотра и упорядочивания.

файлу, где те прописывают каждый раз Version Database — репозиторий.

новые стадии, не отражающие предыдущих. Недостаток в сложности обмена

Сервер занимается построением моделей. контентом между пользователями:

Эта система аналогична структуре “клиент- частая синхронизация VSC и Local

сервер”. Преимущество данной системы репозиториев. А предварительно

в простоте её реализации. Недостаток в необходимо “залить” файл на Local A

одностороннем доступе: Comp B обновляет (если инициатор обмена Comp A).

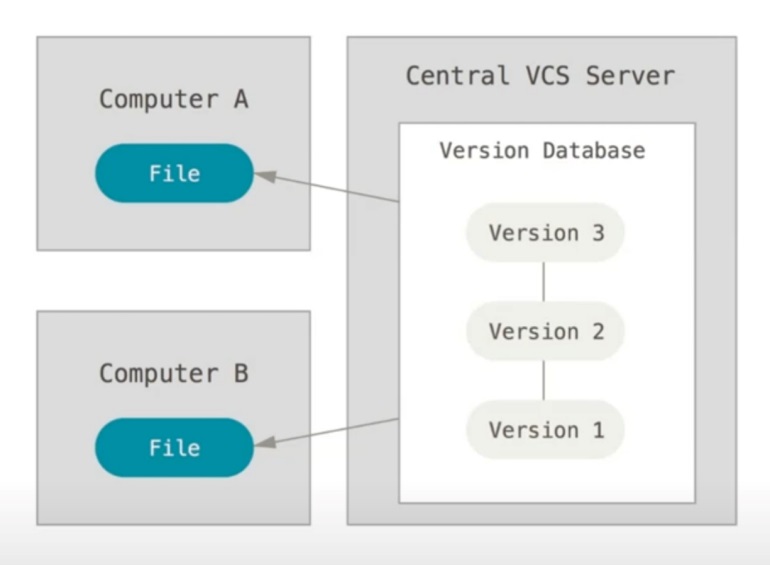
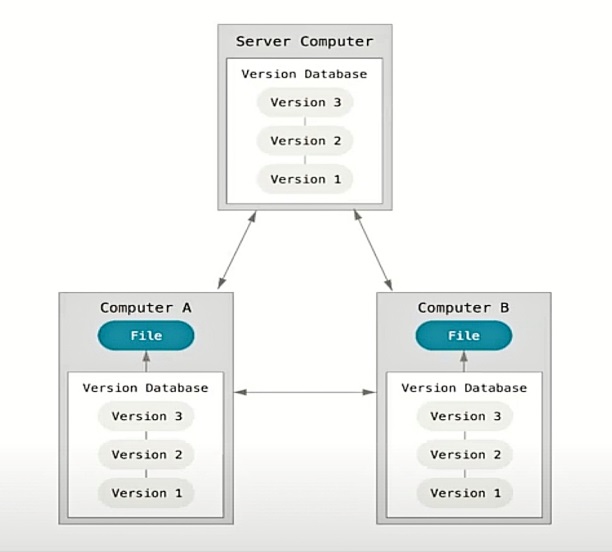
контент из новой версии, внесённой Преимущество в скорости доступа к

Comp A в историю VCS (низкая скорость) . версиям из локального репозитория.

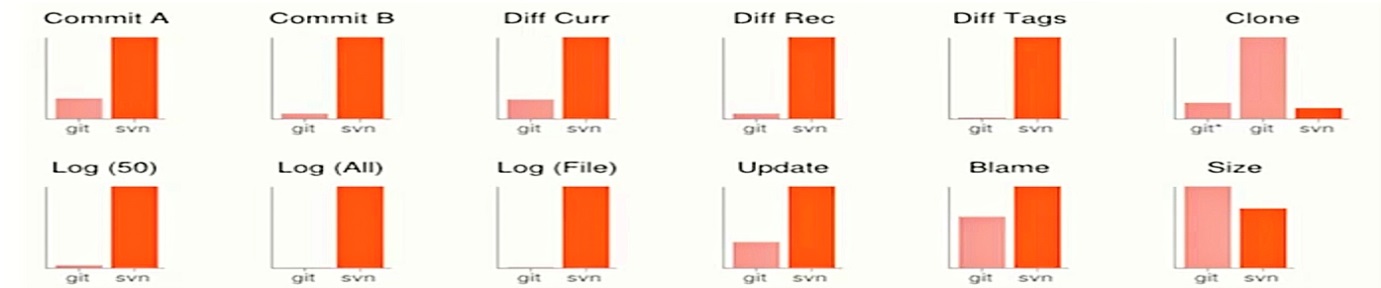
Для сохранности данных нужно создавать Система развёрнутая в сети (облако)

backup(ы). Возрастает нагрузка на сервер медленнее, чем на HDD или SSD диске.

при обращении большего числа user(ов). Фактически это система “сервер-сервер” (снижение работоспособности)

Поговорим теперь непосредственно о GIT. Данная VCS построена на Command Line интерфейсе. GIT является полностью OpenScource(ным), т.е. это продукт с открытым исходным кодом. На текущий момент GIT занимает лидирующее положение по количеству пользующихся им девелоперов (88.4%). Такая популярность обусловлена тем фактом, что GIT выигрывает все VCS по быстродействию выполнения почти всех операций. Так же GIT использует особую систему “Дейташёрнз”, которая обеспечивает полную сохранность данных. Для сохранения файла GIT snapshot(ает) (подвергает контент файла односторонней хеш-функции SHA-1) его. Так же можно говорить о том, что структура GIT подобна графовой: в ней можно производить ветвление, создавать иерархию изменения файлов по определённым признакам.



**1.2. Модули курса**

1.2.1. Скачивание, установка и настройка.

После успешной установки GIT Bash и запуска приложения, перед работой в нём необходимо его сконфигурировать (configure). Сперва эта настройка начинается с генерации т.н. SSH-keys (ключей). Это необходимо для авторизации на таких хостингах как GitHub, GitBucket и т.д.

$ ssh-keygen –t rsa –C “crusxsader@gmail.com” (rsa – алгоритм шифрования; -t, -C — флаги).

Далее нам необходимо разместить ключи (указать путь) и ввести пароль ключей (пароль нужно будет вводить при операциях “подтягивания” (pull) и “забрасывании на сервер” (commit)). В директории .ssh будут хранится два ключа: публичный и приватный.

Далее необходимо загрузить ключи (public-ключ) в тот хостинг, куда будут сохраняться изменения.

После этого настраиваем имя пользователя и адрес эл. почты в Git Bash (это нужно для обозначения авторства некоторых операций).

$ git config --global user.name “Fpm.zhukovic”

$ git config --global user.email “crusxsader@gmail.com”

1.2.2. Создание репозиториев в GitHub и их локальное клонирование.

Инициализируем репозиторий с файлом Readme.md (добавит новый commit) в GitHub.

Клонируем репозиторий на локальную машину (это можно производить либо через расширение протокола HTTP (HTTPS) для поддержки шифрования, либо через SSH ключи). Переходим в директорию, куда разместится проект.

$ cd (change dir) /folder path/

$ pwd (display current working directory)

$ git clone [git@/hosting/.com:/user.name///repo](mailto:git@/hosting/.com:/user.name///repo) name/.git (cloning of repo in dir)

Разрешение на подключение к хосту (в нашем случае github.com), в этом же сообщении от Git Bash мы увидим наш IP-адрес, куда будет привязан код RSA ключа (в кодировке SHA256). Даём согласие (yes). Вводим пароль RSA ключа.

$ ls (checking of creating of dir)

$ cd /name of cloning repo// (appearing of flag: (master) – main branch)

$ git remote -v (ensuring that we work in github: we got mess. about push/fetch origins)

Теперь создадим файл и поместим туда некую информацию. Для простоты будем работать с текстовым документом. Создадим в папке с только что загруженным репозиторием файл song.txt (1-ая строка: sword and spear, 2-ая: wall of shields) и сохраним его.

$ git status (general info about file) (song.txt)

Но, чтобы отправить изменения на хостинг (commit), нам необходимо следовать рекомендации Git Bash (почти после каждой команды система даёт совет, как именно поступить в данной ситуации)

$ git add song.txt (adding file to commit)

$ git status (new file: song.txt)

$ git commit –m “add first 2 lines in the song” (Git Bash message: master [d2d843c] add first 2 lines in the song; 1 changes, 2 insertions(+))

$ git log (history of commits)

(Git Bash message1: commit d2d843c… (HEAD -> master); Author /username. usermail/; Date/day of week. month. day. time. year/); add first 2 lines in the song.

Git Bash message2: commit 73b39ae1… (origin/master, origin/HEAD); Author /username. usermail/; Date; Initial commit.)

$ git push

commit 73b39ae1… — тот, который появился при создании репозитория на хостинге, тут добавлялся Readme.md.

commit d2d843c… — тот, в котором было добавлено 2 строчки в песню

git push — синхронизирует локальный и удалённый репозитории, чтобы изменения вступили в силу и на самом хостинге. Придётся вводить пароль ключа.

Возвращаемся на GitHub и refresh(им) страницу. На сервисе в репозитории появился второй commit и он совпадает с локальным.

1.2.3. Подтягивание изменений с удалённого репозитория

Пусть кто-то создал свой commit и залил его себе на GitHub, а мы хотим его приобрести оттуда.

С помощью опции на GitHub “Edit this file”, что позволяет прямо в интерфейсе GitHub править исходный код и пересылать изменения прямо оттуда, мы допишем ещё 2 строчки (3-ая строчка: standing string; 4-ая строчка: on this their chosen battle field). Commit mess.: add two more lines to the song.

В системе GitHub 3 commits, в Git Bush — 2 commits. ($ git log)

$ git fetch (synchronization of two repo without adding of new content of file locally)

В этом случае нам понадобится команда: $ git merge

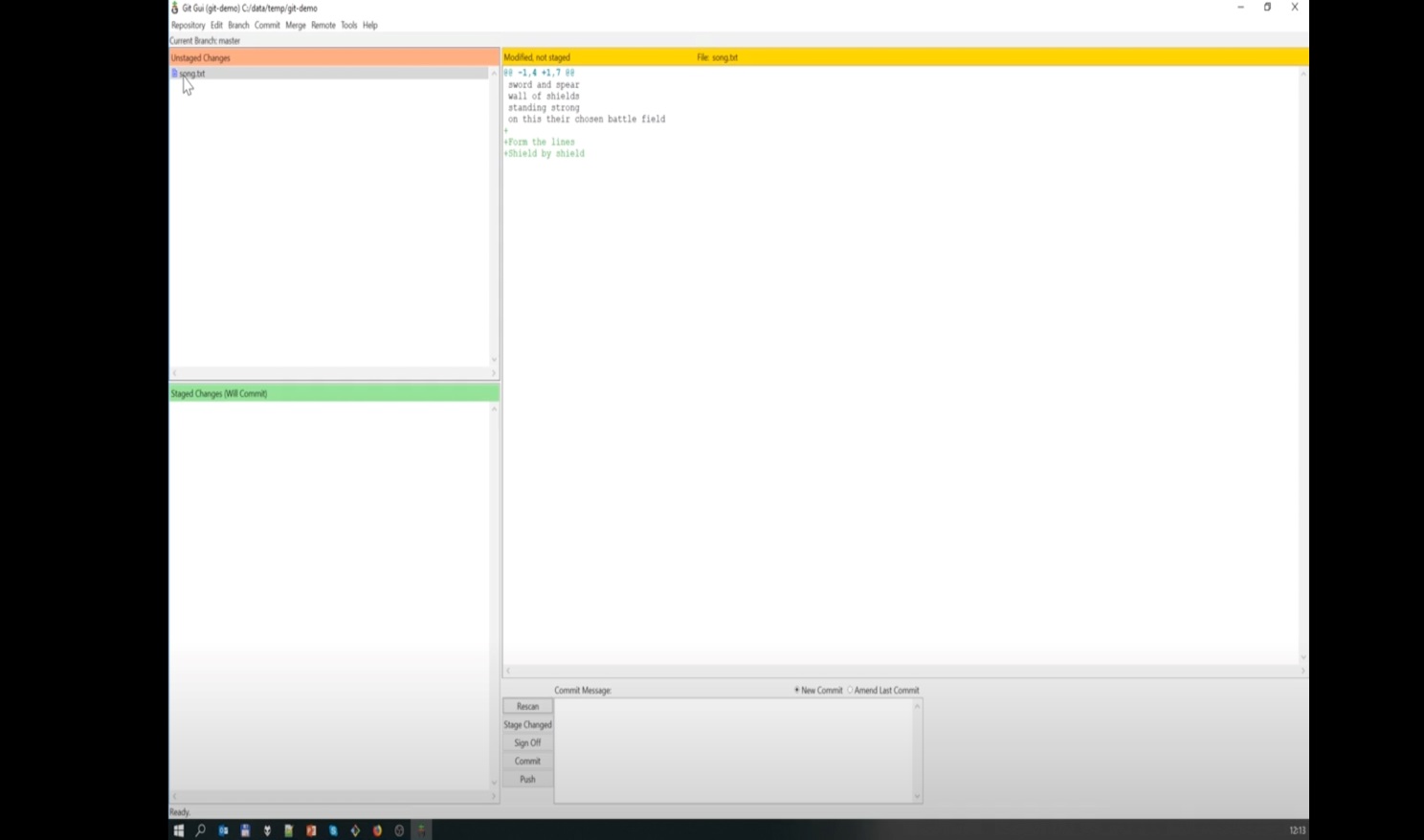
Поэтому целесообразнее: $ git pull (combination of two previous comands) + пароль ключа. (Git Bash message: condition of Enumerating, Counting, Compressing and Unpacking objects; d2d843c…63b9be1 master -> origin/master; Updating d2d843c…; Fast forward (*type of merge*); song.txt | 4 +++-; 1 file changed, 3 insertions(+), 1 deletion(-)).

Локальный файл, в котором изначально было 2 строчки, после reload(а) превратился в полноценную строфу.

1.2.4. Инструменты Git Gui и gitk.

Данные надстройки Git Bush помогают упростить работу в самой системе версионного контроля. Работа в консоле Git Bash оправдана только в случае небольших изменений и правок (например в файл config) или быстрого внесения данных корректив.

Запуск первого инструмента: $ git gui& (& “bridge” between GUI and Git Bush)

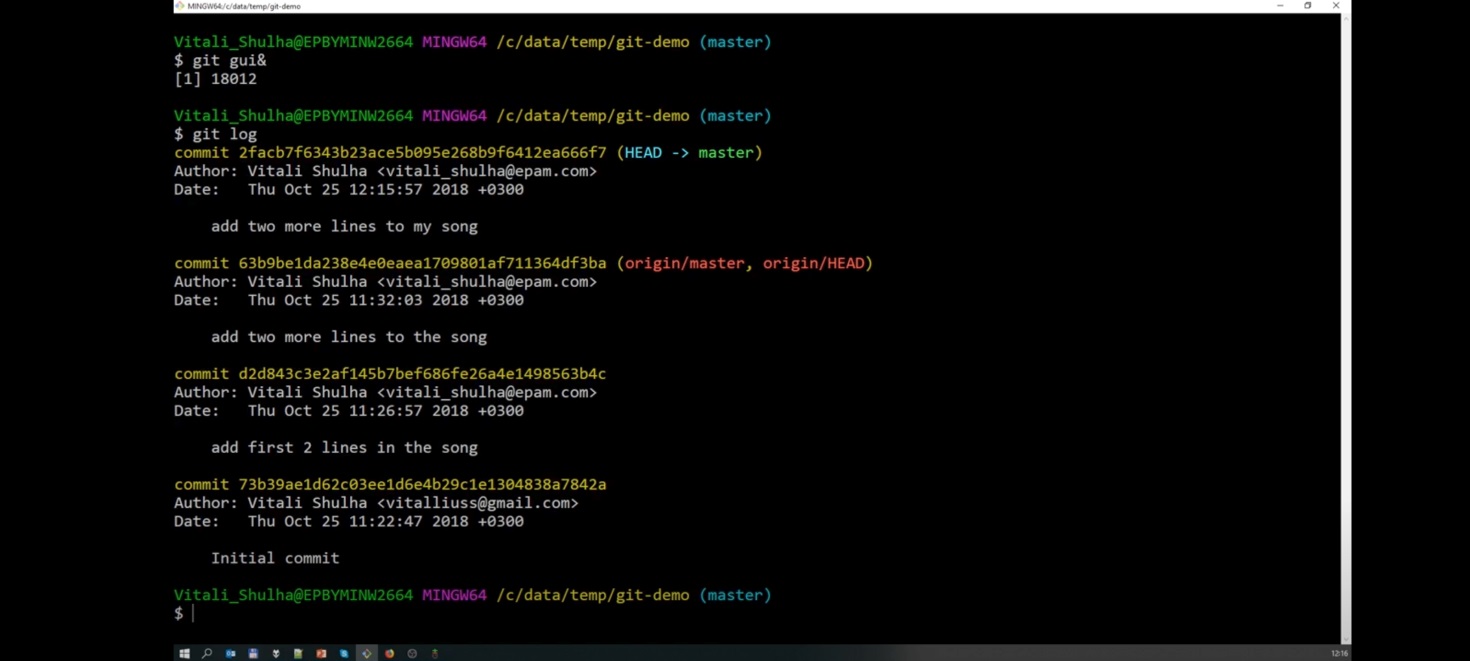


Rescan — синхронизирует состояние файлов, для просмотра поправок в файле

(Предварительно в файл мы добавили пустую строку и две новые строки рифмы).

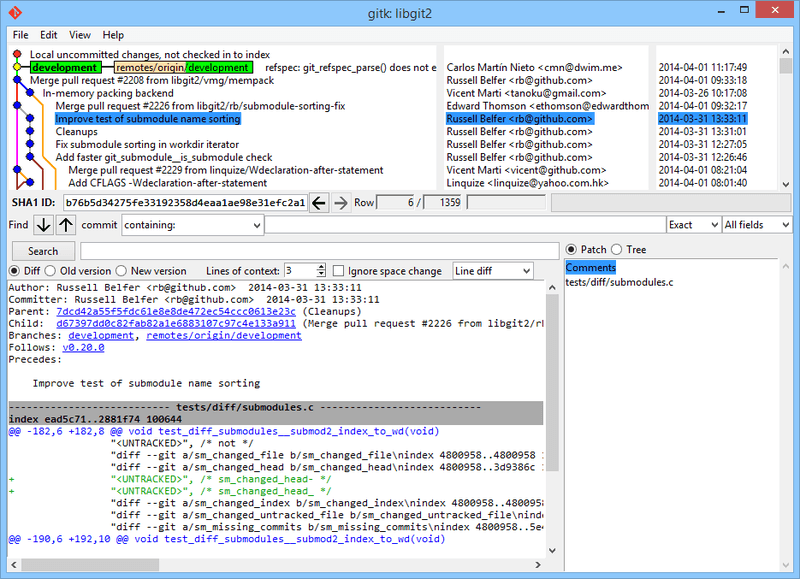
Зелёным (+) отмечается то, что добавилось, а красным (-) — что было удалено. Изменения пока не будут внесены в commit, т.к. находятся в поле Unstaged. Кликом на фиолетовую иконку song.txt мы перемещаем файл в блок Staged (папку Index), который разрешает commit.

Модифицируем дополнительно Readme.md (# git-demo (изначально) + some content). После rescan добавился Readme. В label Commit message пишем комментарий. Кнопка Commit. Вернёмся в Git Bash и с помощью $ git log видим внесённый commit в самом верху.



С помощью кнопки Push (с предоставлением пароля ключа) мы заливаем commit на сервис GitHub.

$ gitk& (данный инструмент предназначен для просмотра истории, отображении SHA-1 кода каждого commit(а), выявления авторства и даты создания каждой модификации, поиска commit(ов) с помощью опции contains, прослеживания иерархии связей между commit(ами), наследования и т.д.).



1.2.5. Хранение информации в GIT.

В GIT всё построено на графах. С помощью SHA-1 GIT обрабатывает файлы. С его помощью система идентифицирует файл, а так же генерирует некий кодовую строку, в зависимости от содержания, где отображён хеш конкретного объекта.

**Hooks** — место размещения скриптов

**Objects** — те данные, которые мы commit(аем) в GIT, хранятся в директориях оттуда.

**HEAD** — тут записана ссылка на некий объект в директории refs/head/master

**refs** — состоит из директорий heads, remotes, tags. В heads расположен SHA-1 ключ к ветке master (40 цифр 16-ричной системы счисления). Ссылка (головной указатель) обращается к последнему (верхнему commit(у)). В remotes содержатся те адреса и состояния commit(ов), которые находятся на remote-репозиториях. После действия команд fetch и merge remote адрес стал таким как в heads (мы уравняли состояние remote и local репозиториев).

**index** — содержит промежуточную информацию, которую мы собираемся commit(ать) (помещаем изменения сюда с помощью add).

**config** — этот файл содержит настройки для вашего репозитория, здесь, например, хранится url вашего репозитория в хранилище, ваше имя, email и так далее.

**info-exclude** — здесь описываются файлы, которые вы не хотите включать в репозиторий. Функционал этого файла такой же, как у файла .gitignore, за исключением того, что он не передается в репозиторий.

**description** — используется gitweb для отображения описания репозитория.

Каждый раз, когда вы создаете файл и коммитите изменения, Git архивирует файл и сохраняет его в своей структуре данных. Архивированный объект создается с уникальным именем и хранится в папке объектов.

По факту, когда вы коммитите изменения, Git производит всего два действия:

1. Если файл в рабочей папке не изменялся, он просто добавляет имя сжатого файла (хеш) в снимок.
2. Если файл в рабочей папке изменялся, он сжимает его, помещает в папку объектов и добавляет имя сжатого файла (хеш) в снимок.

Как только снимок сделан, он также архивируется и именуется при помощи хеша, затем помещается в папку объектов.

├── 4c

│ └── f44f1e3fe4fb7f8aa42138c324f63f5ac85828 // hash

├── 86

│ └── 550c31847e518e1927f95991c949fc14efc711 // hash

├── e6

│ └── 9de29bb2d1d6434b8b29ae775ad8c2e48c5391 // hash

├── info // let's ignore that

└── pack // let's ignore that too

## Вот как выглядит папка объектов после того, как я создал файл file\_1.txt и закоммитил его. Пожалуйста, учтите, что если хеш вашего файла начинается на «4cf44f1e…», то Git сохранит его с именем «f44f1e…» в подпапке с именем «4c». Этот маленький трюк в 255 раз уменьшает количество файлов в папке /objects. Сам коммит состоит из 4-ёх частей: хеш снимка рабочей директории, комментарии, автор коммита, хеш родительского коммита.

Посмотрите сами, что произойдет, если распаковать файл коммита И вот, что вы увидите:

$ git cat-file -p 4cf44f1e3fe4fb7f8aa42138c324f63f5ac85828

tree 86550c31847e518e1927f95991c949fc14efc711 //хеш снимка

author Fpm.zhukovic

<test[@gmail.com](mailto:crusxsader@gmail.com)> 1455775173

-0500

committer Fpm.zhukovic

<[test@gmail.com](mailto:crusxsader@gmail.com)> 1455775173

-0500

commit A

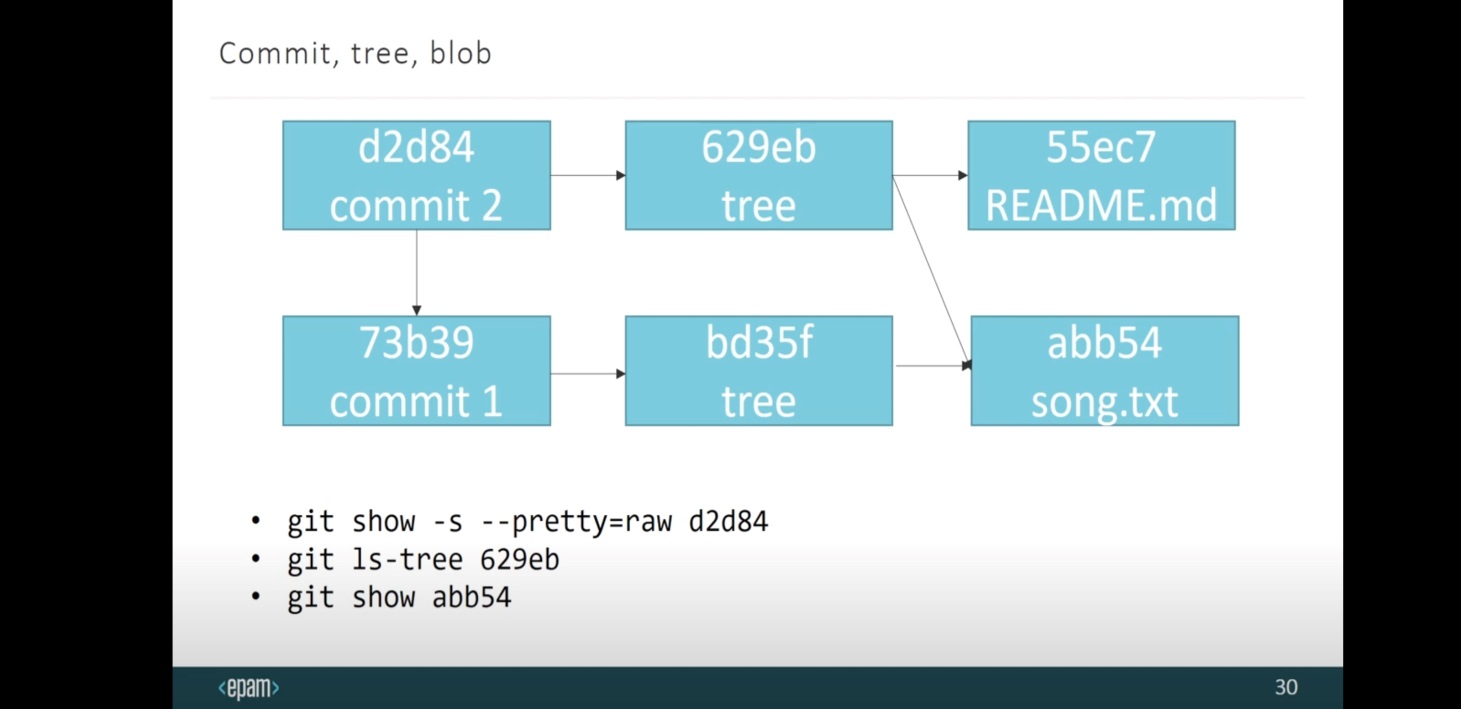
Теперь поговорим об объектах в GIT. Подразделяют их на три группы: commits, tree, blobs (binary large object – минимальная ячейка хранения информации в Git). Commit ссылается на деревья (в них имена файлов с commits и пути к ним) и одновременно на blob (commit1 -> abb54) (дерево так же ссылается на blob, причём несколько tree на один blob). Каждый commit ссылается на предыдущий.

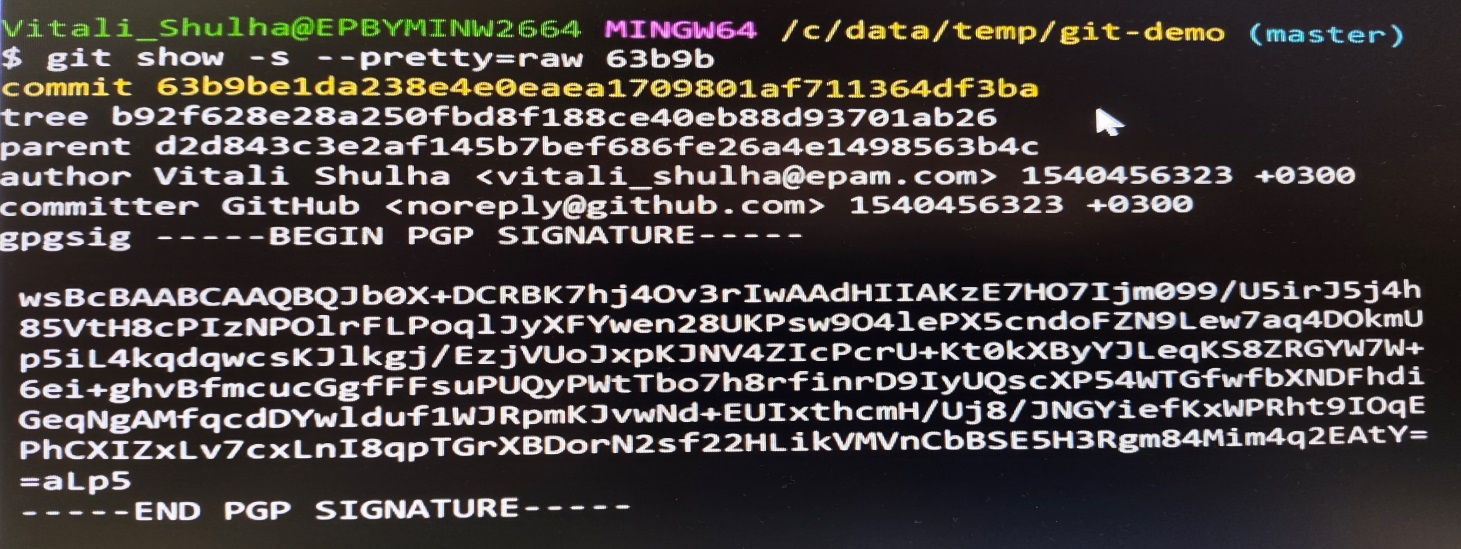
$ git show -s --pretty=raw 63b9b (information about commit)

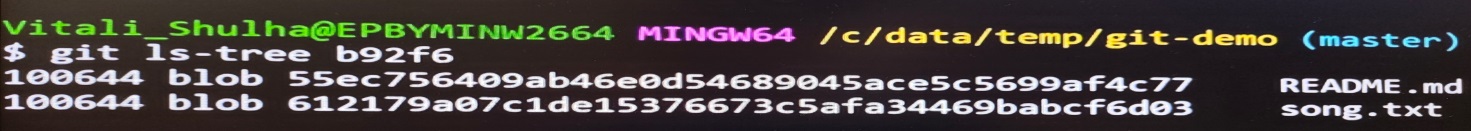
$ git ls-tree b92f6 (information about tree)

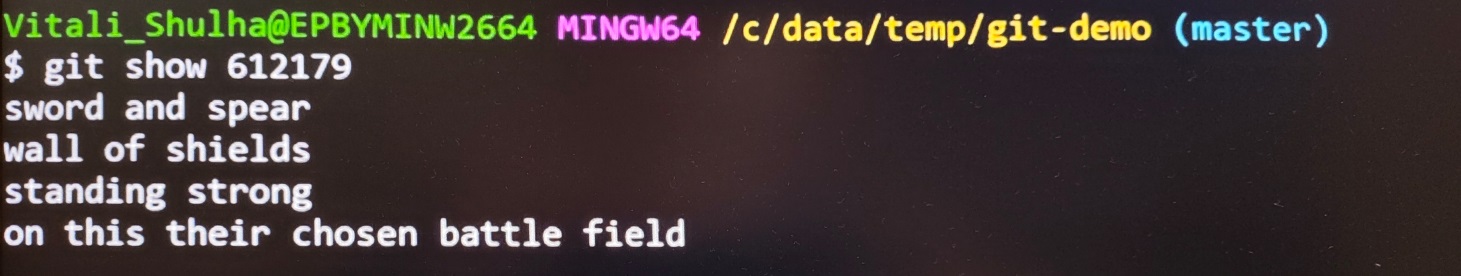
$ git show 612179 (information about blobs)

## Таким образом в Git Bush мы можем “распаковать” commit вплоть до текстового содержимого изменённого файла, что довольно удобно для понимания. Ну и в самом фолдере Objects папки с blobs представлены как двухсимвольные названия (в 16-ти ричной системе) — два символа представляют первые два разряда SHA-1 кода. Это снижает нагрузку на файловую систему и упрощает поиск среди других комитов.









1.2.6. Отмена изменений в GIT.

Необходимо понять на каком уровне находятся изменения, т.к. в соответствии с этим GIT будет предлагать тот или иной способ решения этой проблемы

1) Изменения в файловой системе, которые пока не были внесены в index или commit. Такое часто встречается в программировании, при изменении концепции.

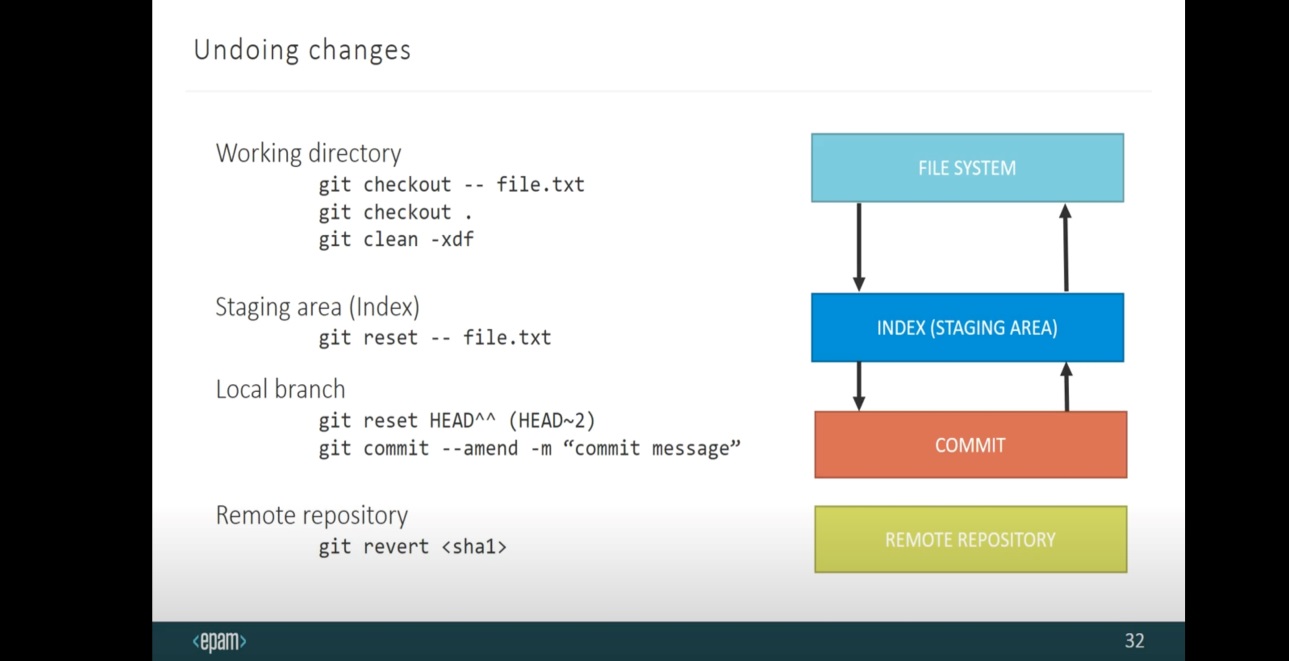
2) Файл создан, модифицирован и добавлен в index. В этом случае файл нужно извлечь из индекса и откатывать после этого.

3) Помимо создания, внесения поправок и добавления в индекс, мы создали commit. Ситуация в этом случае может развернуться так:

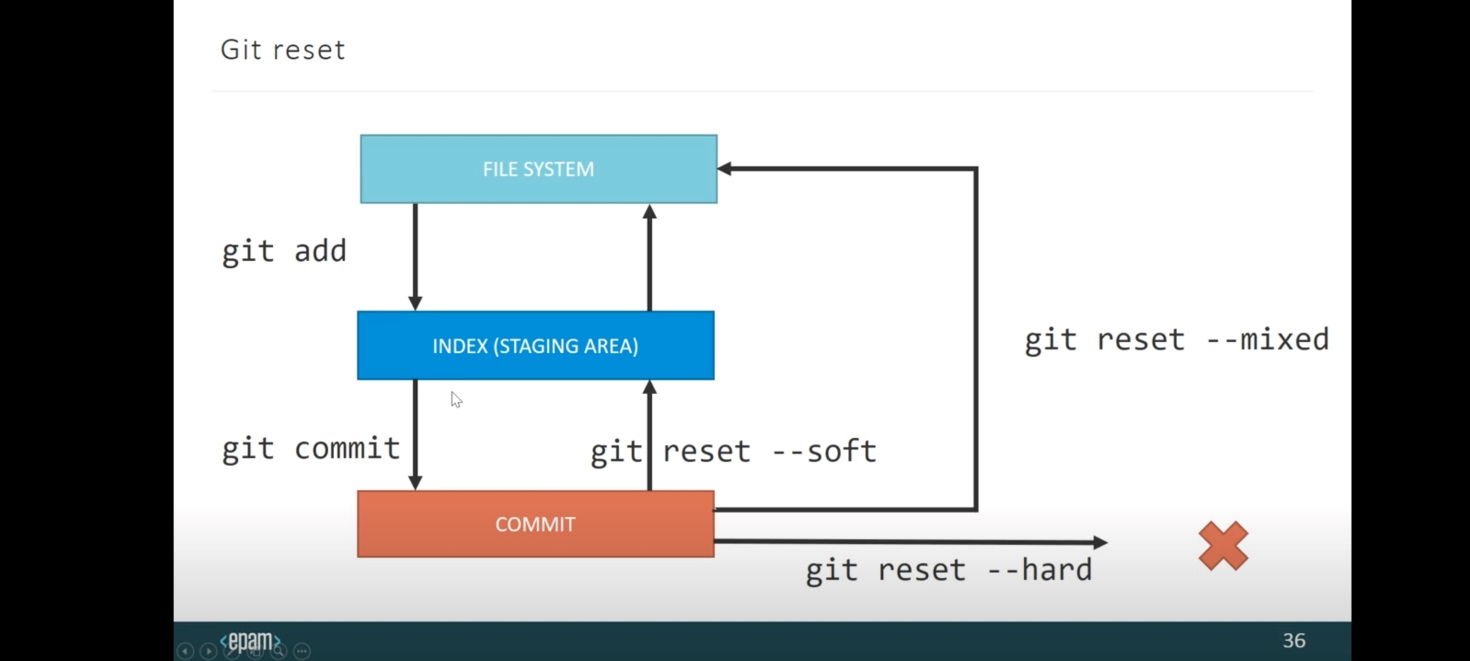
3.1) Если нужно что-то дописать в commit и не удалять старые изменения

3.2) Если нужно всё целиком удалить

4) Удаление репозитория с некоторого хостинга.



1.2.7. Команда Git Reset и откат изменений.



1.2.8. Git Revert и отмена изменений pushed on GitHub.

Пусть мы довершили 2-ой куплет песни и дописали ненужную строку. Мы добавили изменения в индекс, создали commit, залили на GitHub. Теперь наша задача на этом этапе произвести откат. С помощью $ git log узнаём SHA-1 ключ нашего последнего commit(а). $ git revert 06026.

С этого момента мы переносимся в текстовый редактор для режима правок. В этом редакторе мы должны создать commit, который будет осуществлять обратные изменения (добавлять удаляемое, удалять добавляемое).

Для редактирования в редакторе V необходимо нажать insert. Пусть мы вписали некую строчку контента. Теперь перейдём в режим команд (Esc).

:wq – записываем контент файла на диск и выходим из V.

Выйдя из редактора в Git Bash, через $ git log выясняем, что был создан новый реверсионный commit (e86a46…), где так же упомянуты изменения, внесённые в редакторе V.

Произведём push и посмотрим через веб-интерфейс на новые коррективы.