*Модуль 1 (1 пара)*

**Система управления версия(**VCS**)**

1. Введение. История
2. Базовые операции и принципы работы с VCS
3. Словарь терминов
4. Обзор существующих VCS

**Необходимое время – 1 пара.**

Как можно откатиться в текстовом документе? Ctrl+Z. Можно сохранять много рабочих копий одного файла, указывая в имени файла дату, какую-то метку проекта, вести отдельный файл с кратким описанием изменений в файле и содержать в этом файле всю необходимую информацию о списке рабочих файлов и сделанных в них изменениях.

Но это неудобно, проект растет, папка с проектом «пухнет».

А если количество одновременных проектов больше одного? По каждому нужно вести учет об изменениях?

Здесь приходит на помощь VCS.

**WIKI: VCS (version control system)** – ПО для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.

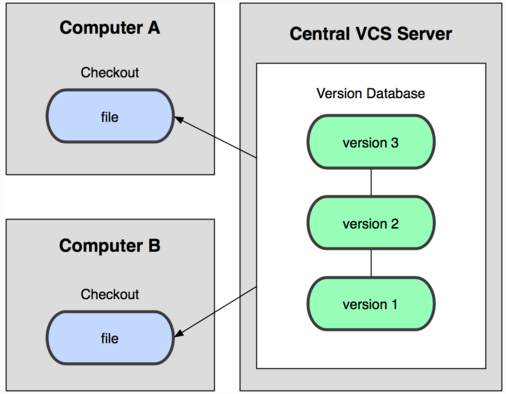
То есть нужно иметь возможно откатиться к более ранним изменениям (snapshots).

Все VCS – по сути хранилище (***repository***), которое хранит ВСЕ версии файлов, с которыми мы работаем. Понятие «версия» - определяется нами, пользователями, теми, кто изменяет документ. К примеру, сделали много мелких правок в документе, «нажали Ctrl-S» – вот вам и получилась «версия». Снова внесли правки – «нажали Ctrl-S» – ЧТО БУДЕТ? Файл перезапишется. Но в случае с VCS, второе «сохранение» не затрется- это будет новая отдельная версия. Очень похоже на игру с сохранением. Поиграли, сохранились, еще поиграли – сохранились. В любой момент мы можем выбрать удобный для нас «сейв» и играть начиная с него, затем снова сохранится. И уже новое сохранение будет считаться новой веткой (***branch).***

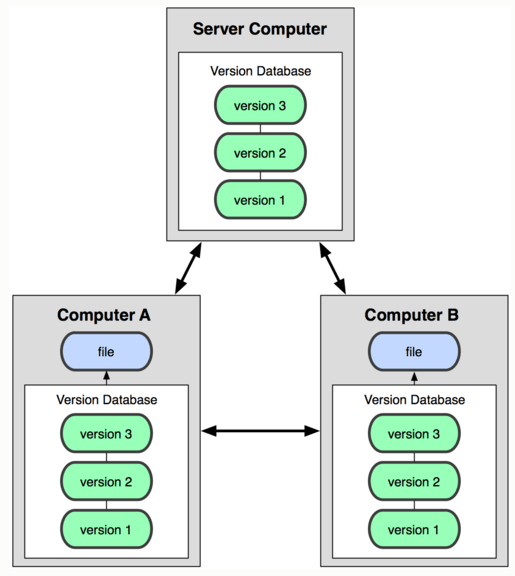
Итак, VCS системы. Данные системы бывают централизованными и распределенными.

Традиционные системы управления версиями используют **централизованную** модель, когда имеется единое хранилище документов, управляемое специальным [сервером](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B2%D0%B5%D1%80_(%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5)), который и выполняет бо́льшую часть функций по управлению версиями.

Пользователь, работающий с документами, должен сначала получить (сделать ***pull)*** нужную ему версию (то есть ***commit)*** документа из хранилища, чаще всего это будет самый последний commit, то есть последняя версия; т.о. создаётся локальная копия документа, так называемая «рабочая копия». Может быть получена последняя версия или любая из предыдущих, которая может быть выбрана по номеру версии или дате создания, иногда и по другим признакам. После того, как в документ внесены нужные изменения, новая версия помещается в хранилище. В отличие от простого сохранения файла, предыдущая версия не стирается, а тоже остаётся в хранилище и может быть оттуда получена в любое время. Все правки, которые мы хотим обозначить новой версией, то есть сделать ***commit***, «улетают» на сервер с какой-то задержкой.



**Распределенные** системы – при получении с сервера некоторой версии, вместо версии мы получаем весь репозиторий проекта.



Поэтому в случае, когда "умирает" сервер, через который шла работа, любой клиентский репозиторий может быть скопирован обратно на сервер, чтобы восстановить базу данных. Каждый раз, когда клиент забирает свежую версию файлов, он создаёт себе полную копию всех данных. Из этого вытекает важное отличие – в распределенных VCS вся работа над проектом ведется с локальным репозиторием. То есть все ***commit***’ы хранятся локально, никуда не улетая, ожидают своего часа. Или если связь с сервером утеряна, всегда можно продолжать работу с локальной копией репозитория.

Примеры VCS: Git, SVN, Perforce и другие **//** *даже Dropbox можно считать системой контроля версий, по сути он и есть VCS.*

Мы познакомимся с vcs git. Она является **распределенной** системой управления версиями.

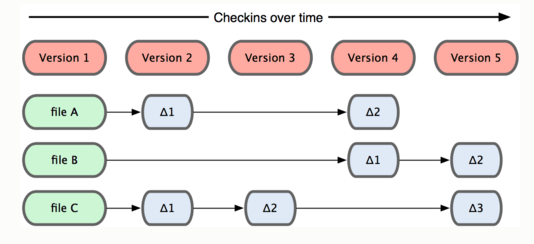
*Модуль 2 (1 пара)*

**Git**

1. Архитектура
2. Установка. Конфигурация.
3. Проверка состояния. Фиксация изменений. Индексация
4. Создание репозитория. Добавление пользователя
5. Добавление файлов. Игнорирование файлов
6. Журнал изменений
7. Понятие ветки. Создание ветки
8. Псевдонимы.
9. **Архитектура**

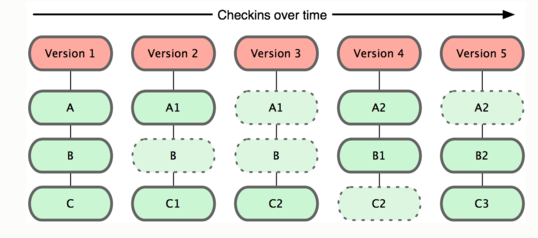
Каким образом можно хранить файлы, подверженные контролю версий?

Можно хранить их «дельты», их еще называют «**патчи**», то есть лишь правки относительно базовой версии:



Так осуществляется контроль в некоторых системах контроля версий.

В Git делается слепок ВСЕГО проекта, а в ***commit*** фиксируется, что НЕизмененные файлы должны иметь ссылку на предыдущую версию (ведь они же не менялись, зачем хранить такую же копию):



Как и упоминалось, все (почти все) операции локальные. Все изменения GIT ведет в локальной копии репозитория (сервер не нагружается). Историю проекта (кто вносил какие изменения) можно увидеть, не обращаясь к серверу. Если вы клонировали себе какой-то репозиторий, помимо всех файлов проекта вы к себе скопировали и всю историю, которая велась над этим проектом. Сразу же вы можете увидеть, когда, что и кем менялось, причем узнавать вы будете это не у сервера VCS, а прямо у себя, у своей копии клонированного репозитория.

1. **Установка. Конфигурация**

**Установка**

На Linux:

***$ yum install git-core – для Fedora (Centos, RHEL)***

***$ apt-get install git – для Debian (Ubuntu)***

На Windows:

<https://git-for-windows.github.io/>

Под Mac тоже есть – гуглим.

**Конфигурация.**

Теперь нужно допилить git под нас. Совсем немного. Открываем GIT BASH и задаем глобально наши имя пользователя и e-mail, то есть повторно их вводить не придется, даже после обновления версии git. Эти данные будут включаться по умолчанию во все коммиты.

*ВНИМАНИЕ. На машинах в академии этот параметр уже задан. Его не изменить, поэтому переходим к следующему пункту.*

***$ git config --global user.name “your name”***

***$ git config --globl user.email “mymail@domain.net”***

Если мы хотим другие указать, к примеру, для другого проекта, то следует перейти в папку с репозиторем и там уже указать другие свои данные, но без ключа ***–global:***

***$ cd anotherProject***

***$ git config user.name “your name”***

***$ git config user.mail*** [***mymail@domain.net***](mailto:mymail@domain.net)

Эта заоверрайдит предыдущие данные, то есть специфичные данные доминируют над глобальными.

Посмотрим на конфиг:

***$ git config --list или***

***$ git config -l***

Если смотреть конфиг, к примеру, из домашнего каталога ~, то увидим глобальные настройки – глобально заданные user.name и user.email. Если перейдем в каталог с созданным и проинициализрованным репозиторием, то увидим и глобальные и специфичные user.name, user.email. Но «съедаться» в коммиты будут последние, то есть специфичные.

**Хелпы.**

Если что-то забыли/не знаете о команде, то есть хелпы:

***$ git help***

***$ git <комманда> -h***

***$ git <комманда> --help***

Или открыть этот же встроенный ***help*** в более читабельном виде на странице:

***$ git <команда> help (из коробки работает только в винде)***

При этом откроется страница в браузере с искомым хелпом. Use it!!

1. **Проверка состояния. Фиксация изменений. Индексация**

**Индексация**

GIT для каждого изменения создает уникальный индекс, делает это он на основании многих переменных (название репозитория, имя измененного файла, автор изменений, сами изменения, время внесенных изменений) – получается на выходе некий хэш, уникальный в своем роде.

Хэш вычисляется по SHA-1 и имеет вид 40-символьного HEX слова:

fcbad4a60ff565adb57319518948e3c4f85797d8

GIT построен таким образом, что окончательно и бесповоротно удалить данные проекта не получится. Если вы удалили что-то в файле (снесли пару строк) и объявили их новой версией (сделали новый ***commit***), потом оказалось, что вы что-то сломали. то GIT просто позволит вернутся к предыдущей еще рабочей копии файла. Причем, делать это он будет ссылаясь на этот самый хэш.

**ВНИМАНИЕ (важная часть)**

Итак, как же работает GIT. В GIT файлы могут находиться в одном из трёх состояний:

1. зафиксированном,
2. изменённом (отмечен красным цветом в GIT BASH)
3. и подготовленном (перед коммитом, отмечены зеленым )

"Зафиксированный" значит, что файл уже сохранён в вашей локальной базе, то есть за***’commit’***ен (закоммитчен). К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы (то есть вы сохранили изменения только у себя на жестком диске, но не в локальном GIT-репозитории). Подготовленные файлы — это изменённые файлы, отмеченные для включения в следующий коммит, но пока не закоммиченные.

Таким образом, в проектах, использующих Git, есть три части (см рис. ниже): каталог Git'а (Git directory), рабочий каталог (working directory) и область подготовленных файлов (***staging area*** или ***index***).

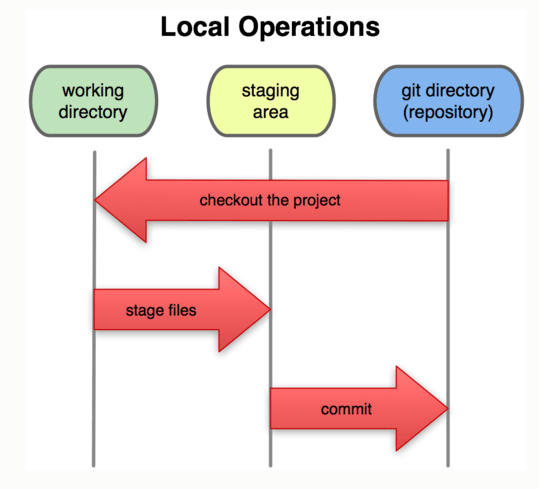
Стандартный рабочий процесс с использованием Git'а выглядит примерно так:

1. Вы вносите изменения в файлы в своём рабочем каталоге и сохраняете их у себя на жестком диске (Ctrl-S).
2. Подготавливаете файлы, добавляя их слепки в область подготовленных (добавляете их в ***index***) файлов, введя команду:

***git add***

1. Делаете коммит, который берёт подготовленные файлы из индекса ***(staging area***) и помещает их в репозиторий Git'а на постоянное хранение:

***git commit –m “any commit description”***



1. **Создание репозитория. Добавление пользователя**

Способ 1. Создать репозиторий в текущем каталоге с проектом. К примеру, вы чего-то накодили, а потом решили подключить git в работу. Находясь в папке со своим проектом (для проекта лучше создать отдельную папку со всем содержимым)

***$ git init***

Вуаля! При этом автоматически создается скрытый служебный каталог ***.git*** (тут, в этом скрытом каталоге, как раз и хранится вся магия git).

Можно добавить свои ***user.name*** и ***user.email***, специфичные для данного проекта.

Rep (репозиторий) готов. И этот реп еще ничего не умеет делать. Теперь добавим под версионный контроль рабочие файлы:

***$ git add <filename>***

***$ git add myproject.php***

Всё. Теперь добавленные файлы git начинает отслеживать, вернее отслеживать их изменения.

Способ 2. (воспользуемся им немного позже) Клонировать существующий rep с удаленного сервера.

***$ git clone <url source> <target folder>***

***$ git clone*** [***https://github.com/superuser/superproject.git***](https://github.com/superuser/superproject.git)

Эта команда создаёт каталог с именем ***superproject***, инициализирует в нём каталог .***git***, скачивает все данные для этого репозитория и создаёт рабочую копию последней версии. Если вы зайдёте в новый каталог ***superproject***, вы увидите в нём проектные файлы, пригодные для работы и использования.

***…или***

***$ git clone*** [***https://github.com/superuser/superproject.git***](https://github.com/superuser/superproject.git) ***<myworkingdirectory>***

Все то же самое, только скопируется rep не в текущую папку, а в папку с названием ***myworkingdirectory.***

Клонировать можно 2 методами по https и по ssh.

1. **Добавление файлов. Игнорирование файлов**

**ВНИМАНИЕ (важная часть)**.

Как добавить файлы под версионный контроль? Выполнить команду:

***$ git add <имя файла>***

Файлы в репозитории находятся в следующих состояниях:

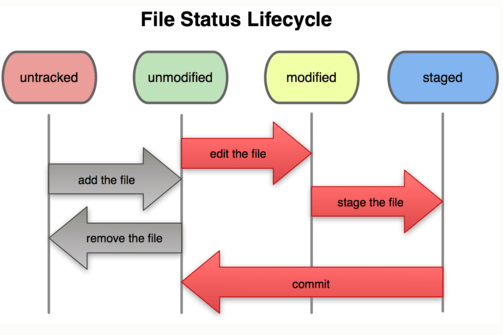
**Untracked** – неотслеживаемые (не добавлены под версионный контроль).

**Tracked** – добавлены под версионный контроль, они же «извлекаются» из последнего коммита, могут быть в следующих состояниях:

**Unmodified** – отслеживаемый файл, из самого «свежего» коммита

**Modified** – отслеживаемый файл, в который мы внесли изменения, нажали Ctrl+S, но не закоммитили или не подготовили к коммиту (то есть, еще не сделали ***git add)***. Отмечен красным цветом в GIT BASH.

**Staged** – отслеживаемый файл является измененным и подготовленным к коммиту (перед коммитом, отмечен зеленым в GIT BASH)



Проверим состояние файлов в репозитории ***git status***:

**$ git status**

On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)

Подобный вывод говорит о том, что рабочий каталог пуст и коммитить нечего. Git также не обнаружил неотслеживаемых файлов, в противном случае они бы были перечислены здесь

On branch master – в какой ветке мы сейчас находимся (по умолчанию это master).

Внесем изменения в проект. Для этого добавим файл:

***$ vi README*** // пишем чего-то, сохраняемся и выходим

Проверим состояние файлов в репозитории

$ git status

On branch master

Initial commit

Untracked files:

(use "git add <file>..." to include in what will be committed)

README

nothing added to commit but untracked files present (use "git add" to track)

Новый файл является untracked, то есть GIT за ним не следит, этого файла не было в предыдущем коммите и GIT не станет его добавлять в ваши будущие коммиты. Для этого GIT нужно «попросить» отслеживать этот файл:

**$ git add README**

**$ git status**

On branch master

Initial commit

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: README

Как видите, файл появился в индексе, то есть в **staged area**, стал зеленым.

**Первый коммит/фиксация изменений**

Коммит – это промежуточный результат вашей работы. Иными словами, это именно то, за что платят деньги кодеру – за рабочий коммит. Коммит это то, что будет видно всем другим разработчикам, работающим с вашими файлами, поэтому следует быть аккуратным ☺

Внесли правки, теперь можно их фиксировать, делая коммит:

***$ git commit –m “краткое описание коммита”***

… или

***$ git commit***

… после ввести краткое описание коммита.

После этого Вы можете видеть, что коммит вывел вам немного информации о себе: на какую ветку вы выполнили коммит (master), какая контрольная сумма SHA-1 у этого коммита (463dc4f), сколько файлов было изменено, а также статистику по добавленным/удалённым строкам в этом коммите.

Еще одна тонкость. Файл может находится одновременно в двух состояниях и красном и зеленом:

(Демонстрация с проектора)

**Разница между изменениями**

Так же можно посмотреть конкретные изменения в файле с момента последнего коммита

***$ git diff*** - разница между последними изменениями и проиндексированными изменеями. То есть между красным и зеленым файлами.

***$ git diff --cached*** - разница между зеленым файлом (то есть проиндексированным файлом) и последним коммитом. То есть покажет, что войдет в следующий коммит.

*Опционально.* ***git commit –a –m “new commit”*** *–* рассказать о ключе **–a** –коммитит все измененные файлы, то есть не только проиндексированные зеленые, но и разом красные (измененные, но не внесенные в индекс).

**Удаление файлов (вообще из проекта).**

Можно удалить руками (длинный путь):

$ ls

new.js new2.js README ttt.txt wow.html www.txt

root@mailsrv MINGW64 ~/testdir (master)

$ rm wow.html

root@mailsrv MINGW64 ~/testdir (master)

$ git status

On branch master

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

deleted: wow.html

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

root@mailsrv MINGW64 ~/testdir (master)

$ git add wow.html

root@mailsrv MINGW64 ~/testdir (master)

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

deleted: wow.html

… и закоммитить удаление.

А можно удалить файл более коротким способом:

***$ git rm <filename>***:

$ git rm wow.html

rm 'wow.html'

root@mailsrv MINGW64 ~/testdir (master)

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

deleted: wow.html

уже готово к коммиту. Если все же решите вернуть файл, то делаем:

***$ git reset HEAD <filename>***

***$ git checkout -- <filename> (можно объединить обе команды через &&)***

**Удаление файлов (из отслеживания, файл останется на месте).**

***$ git add foo.bar***

alexandr@ubnt32:~/Documents/myrep$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: foo.bar

А потом решили, что ему не место под версионным контролем. Удаляем его из индекса отслеживания:

***$ git rm --cached foo.bar***

$/Documents/myrep$ git status

On branch master

Untracked files:

(use "git add <file>..." to include in what will be committed)

foo.bar

**Переименование файла в репозитории**

***$ git mv lib.dll lib2.dll – переименовать файл lib.dll в lib2.dll***

После этого переименование придется закомитить.

**Отмена изменений.**

Пример 1. Изменить предыдущий коммит.

Если мы напортачили с коммитом, забыли чего-то добавить в индекс, или хотим изменить комментарий к коммиту, или просто не хотим плодить ненужные коммиты, то все что нужно сделать это добавить в индекс всё, что мы забыли добавить (***git add <filename>***) и выполнить команду:

***$ git commit --amend***

Мы попадем в диалог создания коммита и нам нужно будет поправить наш комментарий к коммиту, а добавленный файл попадет в этот же коммит. Короче говоря, количество коммитов останется прежним.

Пример 2. Убрать файл из индекса.

К примеру, мы добавили не тот файл для отслеживания (индексации). Как удалить его из отслеживания. Сам GIT и подскажет (***git reset HEAD***):

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: www.txt

new file: www2.txt

Можно ресетнуть какой-то файл, чтобы убрать его из индекса. Проделаем это:

$ git reset HEAD www2.txt

root@mailsrv MINGW64 ~/testdir (master)

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: www.txt

Untracked files:

(use "git add <file>..." to include in what will be committed)

www2.txt

После этого видно, что файл www2.txt изменен (подсвечен красным цветом) но не находится в индексе, то есть в следующий коммит он не войдет.

Пример 2. Отмена изменений в файле.

К примеру, вы изменили файл, но пока не добавили его в индексе:

$ vi www.txt // внесли изменения в редакотре, сохранились..

$ 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: www.txt

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

…но потом вы решили, что изменения неверны, и править файл самостоятельно корректируя его в редакторе будем долгим процессом. Есть путь короче. И GIT нам сам подсказывает. Можно вернуться к состоянию предыдущего коммита для этого конкретного файла:

$ git checkout -- www.txt

root@mailsrv MINGW64 ~/testdir (master)

$ cat www.txt

1st line

И как видим ниже:

$ git status

On branch master

nothing to commit, working tree clean

изменений не стало. ВНИМАНИЕ, изменения внеслись в ваш сохраненный файл, его состояние вернулось к предыдущему коммиту. То есть все, что вы с ним сделали, сотрется из файла и он будет в том же состоянии, что и в предыдущем коммите.

Того же самого можно добиться при помощи команды

***$ git checkout file1.html file2.css file3.js***

Напоследок. Если ваши изменения закоммичены, то можно с высокой долей вероятности сказать, что ваши данные можно восстановить (разумеется, если вы просто не удалили всю папку с репозиторием).

**Временное переключение между коммитами/ветками**

***$ git checkout <хэш коммита>*** **временно** переключиться на коммит с указанным хешем

***$ git checkout master*** вернуться к последнему коммиту в указанной ветке ***master***

**Игнорирование файлов.**

В каталоге проекта могут находится файлы временные, создаваемые во время компиляции или во время работы текстового редактора в ходе работы с файлами проекта, логи и прочее. Их, разумеется коммитить не нужно, их не нужно даже отслеживать! А еще лучше, чтобы git их даже не видел!!

Есть возможность их проигнорировать, создав специальный файл ***.gitignore***, в котором можно перечислить все «лишние» для git’а файлы. Этот файл воспринимает регулярные выражения (regular expressions), то есть шаблоны.

*\* - любая строка*

*? - любой символ*

*anydir/ – игнорировать каталога anydir и его содержимое.*

*[a-zA-Z0-9] – алфавит и цифры с любым регистром вместо 1 символа.*

*test[0-2].js – будут проигнорированы файлы test0.js, test1.js, test2.js*

*! – знак отрицания.*

В файл .***gitignore*** можно вносить комментарии **#**

Нужно создать этот файл, внести в него «маску» и добавить в индекс, после чего файл начнет «работать», то есть GIT будет игнорировать все, что мы перечислили в этом файле.

Файл .gitignore  - это must have в любом серьезном проекте.

1. **Журнал изменений**

***$ git log***

Отсюда видно, кто, когда и что коммитил. Что – в смысле у нас есть уникальный хэш, по которому можно найти изменения, внесенные тем или иным разработчиком.

Можно посмотреть что конкретно было изменено в каждом файле через ключ ***-p***:

***$ git log –p -3***

***-3***  - означает посмотреть последние 3 коммита, и подробности в правках

Вывод истории можно форматировать очень гибко. Тема достаточно велика. В целом того, что есть здесь, достаточно для рутинной работы.

1. **Понятие ветки. Создание ветки**

**Ветки (*branch)***

Ветка – это работа над проектом с его независимой «копией». На самом деле ничего не копируется. Хорошее представление о ветках вот в этой статье:

<https://git-scm.com/book/ru/v1/%D0%92%D0%B5%D1%82%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5-%D0%B2-Git-%D0%A7%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-%D0%B2%D0%B5%D1%82%D0%BA%D0%B0%3F>

Сразу после создания репа, вы находитесь в основной ветке разработки ***master.***

***$ git branch testing -*** cоздать новую ветку

Переключиться на другую существующую ветку:

***$ git checkout testing***  вернуться к последнему коммиту в ветке ***testing***

Можно объединить 2 предыдущие команды одной:

***$ git checkout –b <new branch name>*** - создается новая ветка и тут же в нее переключаемся.

… внесли изменения, проверили, закоммитили в новую ветку. Если все хорошо теперь содержимое ветки ***testing*** можно влить в основную ветку ***master.***

***$ git checkout master -***  перешли в основную ветку

***$ git merge testing –*** влили в нее изменения из ветки ***testing***

Если новая ветка больше не нужна, ее можно удалить. Удалить новую ветку:

***$ git branch -d testing***

Иногда слияние невозможно, поскольку файл в последнем коммите в ветке ***master*** отличается от файла в последнем коммите в ветке ***testing***. После команды ***git merge*** GIT не сольет ветки и остановится, пока не будет решен конфликт слияния.

На помощь придет git status. Она покажет где исправить. Править можно вручную или при помощи графического инструмента

git mergetool

Подробно о конфликтах при слиянии – самостоятельно.

**Переключиться на определенный коммит и продолжить работу с него**

***$ git checkout -b new-branch 5589877*** создать ветку ***new-branch***, начинающуюся с коммита с хэшем ***5589877***

***$ git log* --*graph --all –*** посмотреть все ветки в виде графа.

1. **Псевдонимы**

Сократить ввод команды, обозвав их новыми алиасами.

***$ git config --global alias.co checkout***

***$ git config --global alias.br branch***

***$ git config --global alias.ci commit***

***$ git config --global alias.st status***

*Модуль 3 (1 пара)*

**Github**

1. Регистрация. Обзор интерфейса веб-сервиса.

// *зарегистрироваться, пробежаться по пунктам веб-интерфейса.*

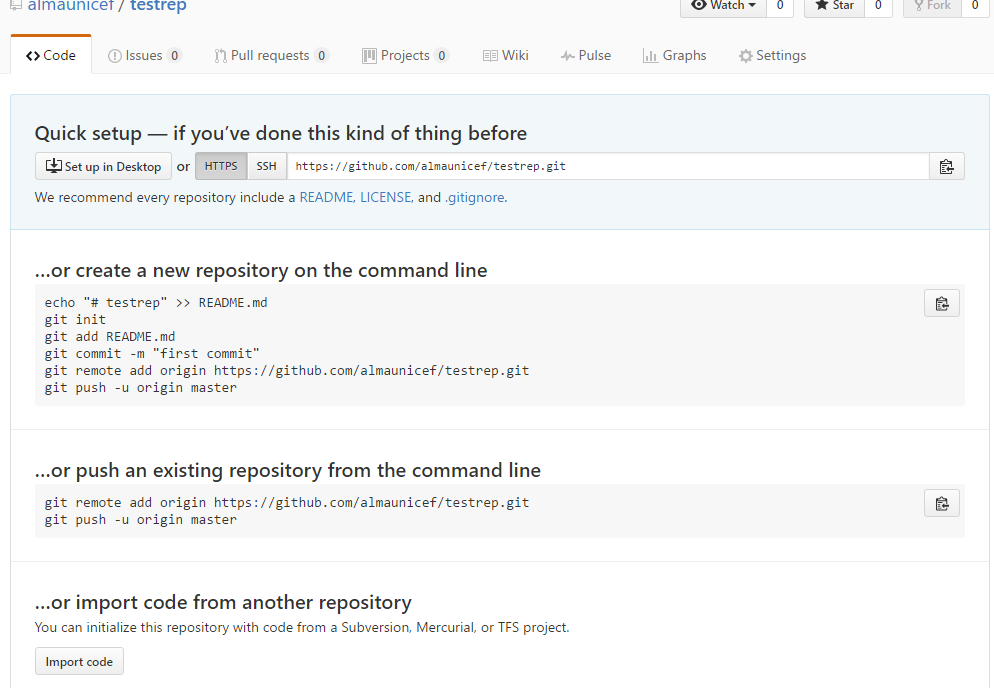
1. Создание удалённого репозитория. Клонирование
2. Использование консоли Git в контексте Github
3. Установка и использование клиента Github
4. Доставка изменений в удалённый репозиторий

**Удаленные репозитории**.

Это те же централизованные/распределенные системы VCS, находящиеся не локально на машине пользователя, а где-то в сети (закрытой или локальной). Github – один из примеров, самый популярный репозиторий, но правда публичный. Можно поднять свой репозиторий на базе GIT.

Для чего нужен удаленный репозиторий? Чтобы делиться коммитами с другими разработчиками или попросту вести работу совместно над каким-то проектом.

Зарегистрируемся, создадим репозиторий, добавим README



Создадим удаленный репозиторий **testrep,** дадим ему description, отредактируем ему README. Клонируем его себе на наш клиент, предварительно создав для него каталог (можно не создавать, во время клонирования создастся папка с именем репозитория):

$ git clone https://github.com/almaunicef/testrep.git

Cloning into 'testrep'...

remote: Counting objects: 3, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0

Unpacking objects: 100% (3/3), done.

Действительно создалась такая папка:

$ ls -l | grep testrep

drwxr-xr-x 1 root 197121 0 фев 9 13:42 testrep/

Перейдем в нее и посмотрим на содержимое:

$ cd testrep/

root@mailsrv MINGW64 ~/testrep (master)

$ ls -l

total 1

-rw-r--r-- 1 root 197121 73 фев 9 13:42 README.md

root@mailsrv MINGW64 ~/testrep (master)

$ cat README.md

# testrep

my 1st student rep

Hello guys!

And "hello world!" too =)

Добавим в веб-интерфейсе файл .gitignore и подтянем его из репозитория себе на клиент. Как подтянуть? Есть такое понятие как  ***pull***. Pull «тянет» что-то откуда-то.

$ git pull origin

Updating fc04038..6c7b1ba

Fast-forward

.gitignore | 2 ++

1 file changed, 2 insertions(+)

create mode 100644 .gitignore

Как видно, к нам подтянулось обновление из репа. Проверим:

$ ls -al

total 14

drwxr-xr-x 1 root 197121 0 фев 9 13:54 ./

drwxr-xr-x 1 root 197121 0 фев 9 13:42 ../

drwxr-xr-x 1 root 197121 0 фев 9 13:54 .git/

-rw-r--r-- 1 root 197121 11 фев 9 13:54 .gitignore

-rw-r--r-- 1 root 197121 73 фев 9 13:42 README.md

root@mailsrv MINGW64 ~/testrep (master)

$ cat .gitignore

\*.log

\*~

При этом, если в вашем репозитории были файлы с такими же именами, к их содержимому добавиться ***merge*** содержимое удаленного репозитория.

Существует еще один метод доставки данных из удаленного репа, ***fetch.*** Нужна если вы хотите получить все ветки удаленного репозитория, но не сливать с нашими ветками.

А теперь мы экспортируем наш готовый репозиторий в обратном направлении, то есть на сервер, но уже в другой реп, к примеру это другой наш проект. Для начала вернемся в нашу директорию с репозиторием:

$ cd ../testdir/

root@mailsrv MINGW64 ~/testdir (master)

$

Теперь нам нужно добавить удаленный репозиторий.

$ git remote add origin https://github.com/almaunicef/testrepv2.git

Эта команда делает следующее: она добавляет URL в качестве удаленного репозитория и дает ему название origin

Посмотрим список удаленных репов:

$ git remote -v

origin https://github.com/almaunicef/testrepv2.git (fetch)

origin https://github.com/almaunicef/testrepv2.git (push)

А теперь отправим в него весь наш проект:

$ git push -u origin master

…при это «вывалится окно с аутентификацией», мы должны подтвердить, что мы являемся владельцем данного удаленного репозитория, чтобы туда что-то отправить

Counting objects: 31, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (23/23), done.

Writing objects: 100% (31/31), 2.57 KiB | 0 bytes/s, done.

Total 31 (delta 9), reused 0 (delta 0)

remote: Resolving deltas: 100% (9/9), done.

Branch master set up to track remote branch master from origin.

To https://github.com/almaunicef/testrepv2.git

\* [new branch] master -> master

Теперь в веб-интерфейсе можно увидеть все наши файлы. И просмотреть детальные изменения для каждого коммита.

*Для подробного изучения git:*

[*https://git-scm.com/book/ru/v1*](https://git-scm.com/book/ru/v1)

<https://github.com/nicothin/web-development/tree/master/git>