

**МИНОБРНАУКИ РОССИИ**

**федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Московский государственный технологический университет «СТАНКИН»**

**(ФГБОУ ВО «МГТУ «СТАНКИН»)**

Институт цифровых интеллектуальных систем  
Кафедра компьютерных систем управления

Образовательная программа 15.03.04

«Автоматизация технологических процессов и производств»

Дисциплина «Основы системного программного обеспечения»

**Лабораторная работа №1**

Тема: «Работа с системами контроля версий на примере Git Hub»

Выполнили:

студенты группы

АДБ 20-06 \_\_\_\_\_\_\_\_\_ Кондратьева Д.Д.

(дата) (подпись)

Принял:

к.т.н., доцент \_\_\_\_\_\_\_\_\_ Ковалев И.А.

(дата) (подпись)

**Москва 2023**

Лабораторная работа №1. Работа с системами контроля версий на примере Git Hub.

Введение

Система контроля версий (СКВ) — это система, регистрирующая изменения в одном или нескольких файлах с тем, чтобы в дальнейшем была возможность вернуться к определённым старым версиям этих файлов. Для примеров будем использовать исходные коды программ, но на самом деле под версионный контроль можно поместить файлы практически любого типа.

Если вы графический или веб-дизайнер и хотели бы хранить каждую версию изображения или макета — а этого вам наверняка хочется — то пользоваться системой контроля версий будет очень мудрым решением. СКВ даёт возможность возвращать отдельные файлы к прежнему виду, возвращать к прежнему состоянию весь проект, просматривать происходящие со временем изменения, определять, кто последним вносил изменения во внезапно переставший работать модуль, кто и когда внёс в код какую-то ошибку, и многое другое. Вообще, если, пользуясь СКВ, вы всё испортите или потеряете файлы, всё можно будет легко восстановить.

Локальные системы контроля версий

Многие предпочитают контролировать версии, просто копируя файлы в другой каталог (как правило, добавляя текущую дату к названию каталога). Такой подход очень распространён, потому что прост, но он и чаще даёт сбои. Очень легко забыть, что ты не в том каталоге, и случайно изменить не тот файл, либо скопировать файлы не туда, куда хотел, и затереть нужные файлы. Чтобы решить эту проблему, программисты уже давно разработали локальные СКВ с простой базой данных, в которой хранятся все изменения нужных файлов (см. рисунок 1).

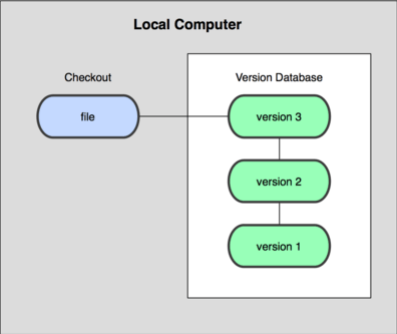


Рисунок 1. Схема локальной СКВ.

Одной из наиболее популярных СКВ такого типа является rcs, которая до сих пор устанавливается на многие компьютеры. Даже в современной операционной системе Mac OS X утилита rcs устанавливается вместе с Developer Tools. Эта утилита основана на работе с наборами патчей между парами версий (патч — файл, описывающий различие между файлами), которые хранятся в специальном формате на диске. Это позволяет пересоздать любой файл на любой момент времени, последовательно накладывая патчи.

Централизованные системы контроля версий

Следующей основной проблемой оказалась необходимость сотрудничать с разработчиками за другими компьютерами. Чтобы решить её, были созданы централизованные системы контроля версий (ЦСКВ). В таких системах, например, CVS, Subversion и Perforce, есть центральный сервер, на котором хранятся все файлы под версионным контролем, и ряд клиентов, которые получают копии файлов из него. Много лет это было стандартом для систем контроля версий (см. рисунок 2).

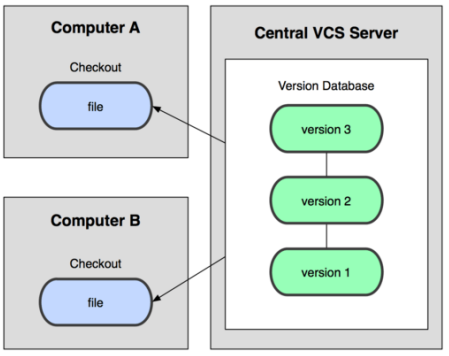


Рисунок 2. Схема централизованного контроля версий.

Такой подход имеет множество преимуществ, особенно над локальными СКВ. К примеру, все знают, кто и чем занимается в проекте. У администраторов есть чёткий контроль над тем, кто и что может делать, и, конечно, администрировать ЦСКВ намного легче, чем локальные базы на каждом клиенте.

Однако при таком подходе есть и несколько серьёзных недостатков. Наиболее очевидный — централизованный сервер является уязвимым местом всей системы. Если сервер выключается на час, то в течение часа разработчики не могут взаимодействовать, и никто не может сохранить новой версии своей работы. Если же повреждается диск с центральной базой данных и нет резервной копии, вы теряете абсолютно всё — всю историю проекта, разве что за исключением нескольких рабочих версий, сохранившихся на рабочих машинах пользователей. Локальные системы контроля версий подвержены той же проблеме: если вся история проекта хранится в одном месте, вы рискуете потерять всё.

Распределённые системы контроля версий

И в этой ситуации в игру вступают распределённые системы контроля версий (РСКВ). В таких системах как Git, Mercurial, Bazaar или Darcs клиенты не просто выгружают последние версии файлов, а полностью копируют весь репозиторий. 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 (см. рисунок 3).

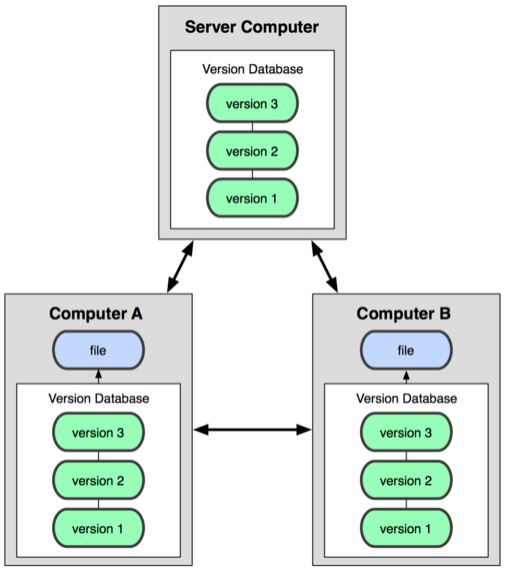


Рисунок 3. Схема распределённой системы контроля версий.

Кроме того, в большей части этих систем можно работать с несколькими удалёнными репозиториями, таким образом, можно одновременно работать по-разному с разными группами людей в рамках одного проекта. Так, в одном проекте можно одновременно вести несколько типов рабочих процессов, что невозможно в централизованных системах.

Краткая история Git

Как и многие замечательные вещи, Git начинался с, в некотором роде, разрушения во имя созидания и жарких споров. Ядро Linux — действительно очень большой открытый проект. Бо́ льшую часть существования ядра Linux (1991-2002) изменения к нему распространялись в виде патчей и заархивированных файлов. В 2002 году проект перешёл на проприетарную РСКВ BitKeeper.

В 2005 году отношения между сообществом разработчиков ядра Linux и компанией, разрабатывавшей BitKeeper, испортились, и право бесплатного пользования продуктом было отменено. Это подтолкнуло разработчиков Linux (и в частности Линуса Торвальдса, создателя Linux) разработать собственную систему, основываясь на опыте, полученном за время использования BitKeeper. Основные требования к новой системе были следующими:

* Скорость
* Простота дизайна
* Поддержка нелинейной разработки (тысячи параллельных веток)
* Полная распределённость
* Возможность эффективной работы с такими большими проектами, как ядро Linux (как по скорости, так и по размеру данных)

С момента рождения в 2005 году Git развивался и эволюционировал, становясь проще и удобнее в использовании, сохраняя при этом свои первоначальные качества. Он невероятно быстр, очень эффективен для больших проектов, а также обладает превосходной системой ветвления для нелинейной разработки.

Основы Git

Так что же такое Git в двух словах? Эту часть важно усвоить, поскольку если вы поймёте, что такое Git, и каковы принципы его работы, вам будет гораздо проще пользоваться им эффективно. Изучая Git, постарайтесь освободиться от всего, что вы знали о других СКВ или постоянно пользуетесь, таких как Subversion или Perforce. В Git'е совсем не такие понятия об информации и работе с ней как в других системах, хотя пользовательский интерфейс очень похож. Знание этих различий защитит вас от путаницы при использовании Git'а.

Слепки вместо патчей

Главное отличие Git'а от любых других СКВ (например, Subversion и ей подобных) — это то, как Git смотрит на свои данные. В принципе, большинство других систем хранит информацию как список изменений (патчей) для файлов. Эти системы (CVS, Subversion, Perforce, Bazaar и другие) относятся к хранимым данным как к набору файлов и изменений, сделанных для каждого из этих файлов во времени, как показано на рисунке 4.

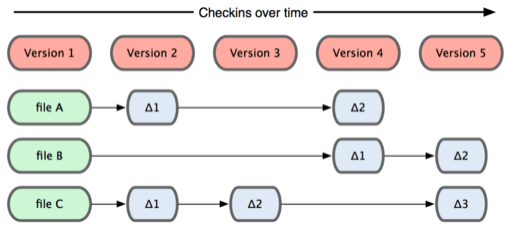


Рисунок 4. Другие системы хранят данные как изменения к базовой версии для каждого файла.

Git не хранит свои данные в таком виде. Вместо этого Git считает хранимые данные набором слепков небольшой файловой системы. Каждый раз, когда вы фиксируете текущую версию проекта, Git, по сути, сохраняет слепок того, как выглядят все файлы проекта на текущий момент. Ради эффективности, если файл не менялся, Git не сохраняет файл снова, а делает ссылку на ранее сохранённый файл. То, как Git подходит к хранению данных, похоже на рисунок 5.

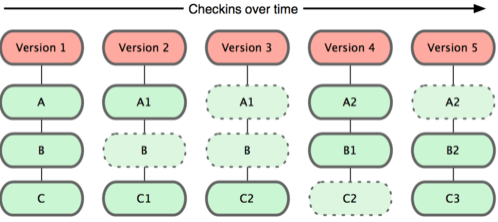


Рисунок 5. Git хранит данные как слепки состояний проекта во времени.

Это важное отличие Git'а от практически всех других систем контроля версий. Из-за него Git вынужден пересмотреть практически все аспекты контроля версий, которые другие системы переняли от своих предшественниц. Git больше похож на небольшую файловую систему с невероятно мощными инструментами, работающими поверх неё, чем на просто СКВ.

Почти все операции — локальные

Для совершения большинства операций в Git'е необходимы только локальные файлы и ресурсы, т.е. обычно информация с других компьютеров в сети не нужна. Если вы пользовались централизованными системами, где практически на каждую операцию накладывается сетевая задержка, вы, возможно, подумаете, что боги наделили Git неземной силой. Поскольку вся история проекта хранится локально у вас на диске, большинство операций кажутся практически мгновенными.

К примеру, чтобы показать историю проекта, Git'у не нужно скачивать её с сервера, он просто читает её прямо из вашего локального репозитория. Поэтому историю вы увидите практически мгновенно. Если вам нужно просмотреть изменения между текущей версией файла и версией, сделанной месяц назад, Git может взять файл месячной давности и вычислить разницу на месте, вместо того чтобы запрашивать разницу у СКВ-сервера или качать с него старую версию файла и делать локальное сравнение.

Кроме того, работа локально означает, что мало чего нельзя сделать без доступа к Сети или VPN. Если вы в самолёте или в поезде и хотите немного поработать, можно спокойно делать коммиты, а затем отправить их, как только станет доступна сеть. Если вы пришли домой, а VPN-клиент не работает, всё равно можно продолжать работать. Во многих других системах это невозможно или же крайне неудобно. Например, используя Perforce, вы мало что можете сделать без соединения с сервером. Работая с Subversion и CVS, вы можете редактировать файлы, но сохранить изменения в вашу базу данных нельзя (потому что она отключена от репозитория). Вроде ничего серьёзного, но потом вы удивитесь, насколько это меняет дело.

Git следит за целостностью данных

Перед сохранением любого файла Git вычисляет контрольную сумму, и она становится индексом этого файла. Поэтому невозможно изменить содержимое файла или каталога так, чтобы Git не узнал об этом. Эта функциональность встроена в сам фундамент Git'а и является важной составляющей его философии. Если информация потеряется при передаче или повредится на диске, Git всегда это выявит.

Механизм, используемый Git'ом для вычисления контрольных сумм, называется SHA-1 хешем. Это строка из 40 шестнадцатеричных символов (0- 9 и a-f), вычисляемая в Git'е на основе содержимого файла или структуры каталога. SHA-1 хеш выглядит примерно так:

B6DABFD326E06D8896CBE83B3BD52645CCE69C28

Работая с Git'ом, вы будете встречать эти хеши повсюду, поскольку он их очень широко использует. Фактически, в своей базе данных Git сохраняет всё не по именам файлов, а по хешам их содержимого.

Чаще всего данные в Git только добавляются

Практически все действия, которые вы совершаете в Git'е, только добавляют данные в базу. Очень сложно заставить систему удалить данные или сделать что-то неотменяемое. Можно, как и в любой другой СКВ, потерять данные, которые вы ещё не сохранили, но как только они зафиксированы, их очень сложно потерять, особенно если вы регулярно отправляете изменения в другой репозиторий.

Поэтому пользоваться Git'ом — удовольствие, потому что можно экспериментировать, не боясь что-то серьёзно поломать.

Три состояния

Теперь внимание. Это самое важное, что нужно помнить про Git, если вы хотите, чтобы дальше изучение шло гладко. В Git'е файлы могут находиться в одном из трёх состояний: зафиксированном, изменённом и подготовленном. "Зафиксированный" значит, что файл уже сохранён в вашей локальной базе. К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы. Подготовленные файлы — это изменённые файлы, отмеченные для включения в следующий коммит.

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

Каталог Git'а — это место, где Git хранит метаданные и базу данных объектов вашего проекта. Это наиболее важная часть Git'а, и именно она копируется, когда вы клонируете репозиторий с другого компьютера.

Рабочий каталог — это извлечённая из базы копия определённой версии проекта. Эти файлы достаются из сжатой базы данных в каталоге Git'а и помещаются на диск для того, чтобы вы их просматривали и редактировали.

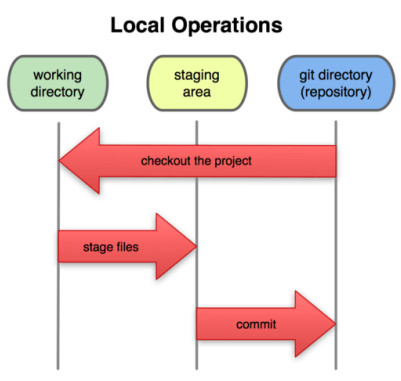


Рисунок 6. Рабочий каталог, область подготовленных файлов, каталог Git'а.

Область подготовленных файлов — это обычный файл, обычно хранящийся в каталоге Git'а, который содержит информацию о том, что должно войти в следующий коммит. Иногда его называют индексом (index), но в последнее время становится стандартом называть его областью подготовленных файлов (staging area).

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

1. Вы вносите изменения в файлы в своём рабочем каталоге.
2. Подготавливаете файлы, добавляя их слепки в область подготовленных файлов.
3. Делаете коммит, который берёт подготовленные файлы из индекса и помещает их в каталог Git'а на постоянное хранение.

Если рабочая версия файла совпадает с версией в каталоге Git'а, файл считается зафиксированным. Если файл изменён, но добавлен в область подготовленных данных, он подготовлен. Если же файл изменился после выгрузки из БД, но не был подготовлен, то он считается изменённым.

# **Индивидуальные задания**

Создали учетную запись на github.com

Зашли на github.com

Зарегистрировались на нем и подтвердили регистрацию на своей почте.

Перед тем, как создавать репозиторий, перешли в Settings и изменили название главной ветки с «main» на «master» и нажали Update.

Дале создали новый репозиторий.

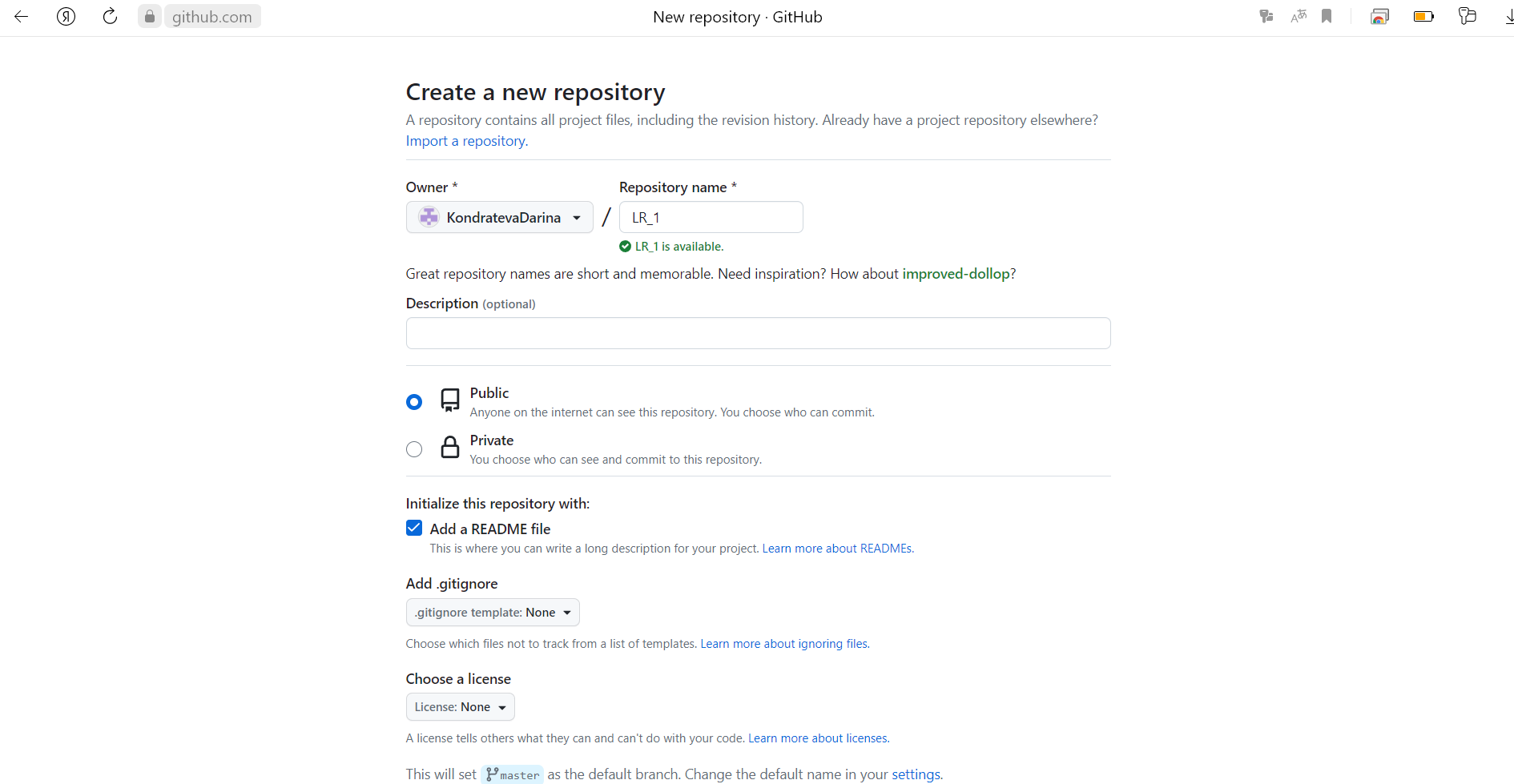


Рисунок 7. Создание репозитория.

Перед нами появилось окно создания нового репозитория.

Также можно установить галочку напротив предложения создать файл README – считается хорошим тоном, если кто-то заходит в ваш репозиторий и первым делом ему будет открываться описание вашего проекта.

Создание локального репозитория

Создадим теперь локальную папку, которая будет связана с нашим облачным git хранилищем. В общем случаем облачный git может представлять также локальный сервер, объединенный в единую информационную сеть со всеми ПК.

Создали на диске папку со своей фамилией. Создали в ней файлик test.txt и написали в нем свою фамилию.

Откройте командную строку «cmd» и перейдите с помощью команды «cd» в созданный вами каталог. Проще всего это сделать, выделив путь в проводнике, написать вместо него cmd и нажать enter.

Запустите команды (со своими данными):

git config --global user.name "My Name"

git config --global user.email myEmail@example.com

Выполнили эти команды, иначе все ваши изменения могут помечаться почтой и именем того, кто раньше представился.



Рисунок 8. Настройка параметров.

Перешли в созданную вами папку в командной строке используя команду cd.



Рисунок 9. Переход в созданную нами папку.

Теперь необходимо проинициализировать эту папку как git репозиторий



Рисунок 10. Инициализация папки как git репозиторий.

Командная строка вернула нам сообщение, что проинициализирована пустая git директория. Теперь используя проводник windows создадим в своей папке любой текстовый файл и пропишем в него своё имя.

Введём команду:

git status

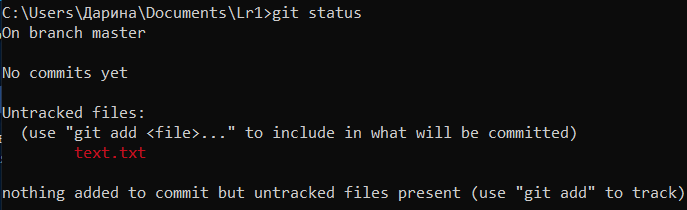


Рисунок 11. Git status.

Нам отобразилось сообщение, что есть новый файл, но он не отслеживается. Это значит, что файл новый и система еще не знает, нужно ли следить за изменениями в файле или его можно просто игнорировать. Для того, чтобы начать отслеживать новый файл, нужно его специальным образом объявить.

В git есть концепция области подготовленных файлов. Можно представить ее как холст, на который наносят изменения, которые нужны в коммите. Сперва он пустой, но затем мы добавляем на него файлы (или части файлов, или даже одиночные строчки) командой add и, наконец, коммитим все нужное в репозиторий (создаем слепок нужного нам состояния) командой commit.

Фиксация изменений в области заготовленных файлов

Если хотим добавить только один файл, можно написать: git add test.txt

Если измененных файлов много, то можно сделать так: git add .

Выполнив нужную команду, снова посмотрим статус репозитория.

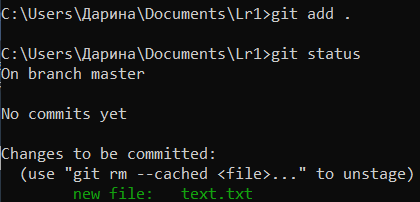


Рисунок 12. Добавление файлов.

Файл готов к коммиту. Сообщение о состоянии также говорит нам о том, какие изменения относительно файла были проведены в области подготовки — в данном случае это новый файл, но файлы могут быть модифицированы или удалены.

Теперь закомитим наши файлы. Коммит представляет собой состояние репозитория в определенный момент времени как некий опечаток во времени. Коммит отмечается hash суммой и к которому мы можем в любой момент времени вернуться. Чтобы зафиксировать изменения, нам нужно хотя бы одно изменение в области подготовки (мы как раз создали новый файл):

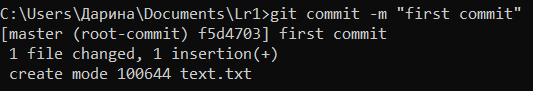


Рисунок 13. Коммит файла.

Команда создала новый коммит со всеми изменениями из области подготовки. Считается хорошей практикой делать коммиты часто и всегда писать содержательные комментарии.



Рисунок 14. Состояние проекта.

Отправка коммита на сервер

Чтобы связать наш локальный репозиторий с репозиторием на GitHub, выполним следующую команду в терминале.

git remote add origin https://github.com/KondratievViktor/LR1



Рисунок 15. Связка локального репозитория с репозиторием на GitHub.

Проект может иметь несколько удаленных репозиториев одновременно. Чтобы их различать, мы дадим им разные имена. Обычно главный репозиторий называется origin. Можно вызвать команду, для просмотра, к какому проекту мы подключены.

git remote -v

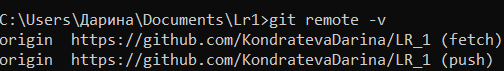


Рисунок 16.Просмотр проекта подключения.

Если вдруг у вас на этом месте показывается несколько репозиториев, то скорее всего верхние команды привели к запутыванию веток, это можно разрешить следующей командой. Также это команда может пригодиться, если у вас возникли ошибки, связанные с unrelated-histories.

git pull origin master --allow-unrelated-histories

Запустите git remote -v снова

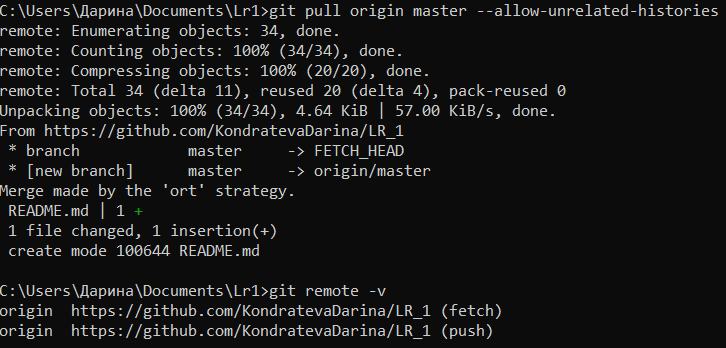


Рисунок 17.

Запросим изменения с сервера

В нашем случае у нас есть файл и на сервере и локальном репозитории, поэтому нам вначале надо получить все данные с нашего репозитория, а потом уже их заливать на сервер нашими локальными данными. Если вы сделали изменения в вашем репозитории (а мы их сделали, создав файл readme), другие пользователи могут скачать изменения при помощи команды pull.

git pull origin master

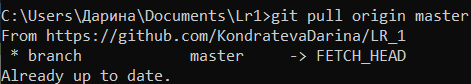


Рисунок 18. Запрос изменения с сервера.

Теперь перешлем локальный коммит на сервер

Теперь отравим коммит на сервер, команда, предназначенная для этого — push. Она принимает два параметра: имя удаленного репозитория (мы назвали наш origin) и ветку, в которую необходимо внести изменения (master — это ветка по умолчанию для всех репозиториев).

git push origin master

У вас должно появиться окно, где нужно ввести авторизоваться на github.com

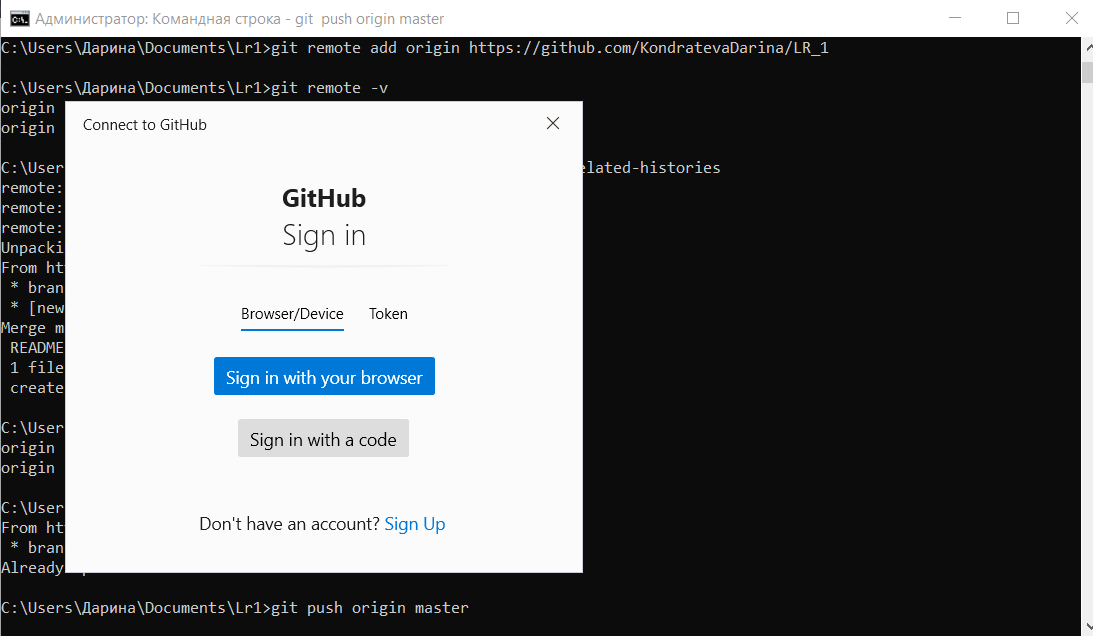


Рисунок 19. Отправка коммита на сервер.

Посмотрим наши изменения с использованием команды

git log

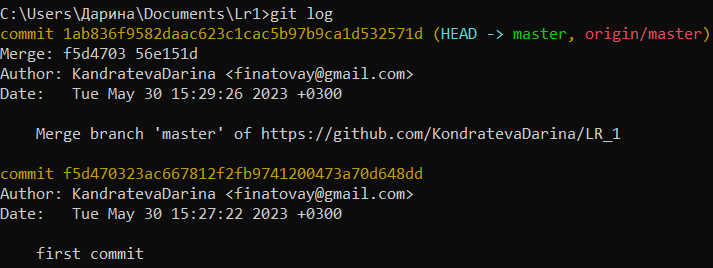


Рисунок 20. Коммиты по изменениям.

Посмотрим изменения на github.com

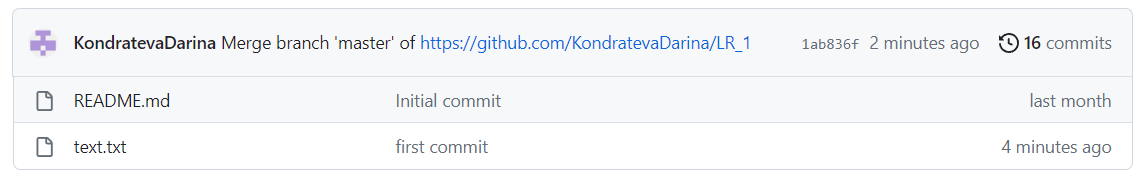


Рисунок 21. Изменения на github.

Если нажать на «commint», то увидим все наши изменения

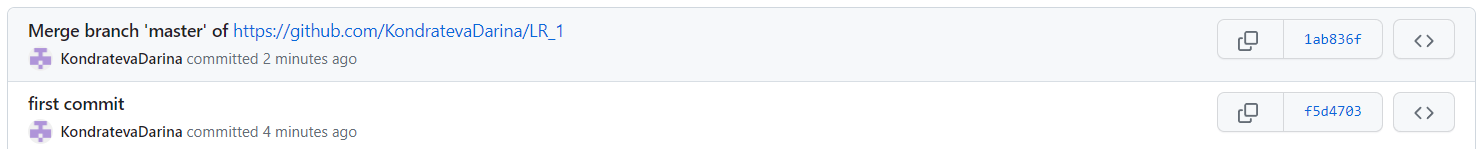


Рисунок 22. Все изменения.

Создание новой ветки

Во время разработки новой функциональности считается хорошей практикой работать с копией оригинального проекта, которую называют веткой. Ветви имеют свою собственную историю и изолированные друг от друга изменения до тех пор, пока вы не решаете слить изменения вместе. Это происходит по набору причин:

* Уже рабочая, стабильная версия кода сохраняется.
* Различные новые функции могут разрабатываться параллельно разными программистами.
* Разработчики могут работать с собственными ветками без риска, что кодовая база поменяется из-за чужих изменений.
* В случае сомнений, различные реализации одной и той же идеи могут быть разработаны в разных ветках и затем сравниваться.

Обычно в проекте создается минимум 2 ветки (branch) – master и ветка для разработки. Ветка master создается сам, а дополнительные ветки создаются по желанию пользователя (локально), заливаются они в удаленный репозиторий уже с учетом доступных прав. Создадим новую ветку second:

git branch second

Можно посмотреть в какой ветки находимся, используя команду git branch

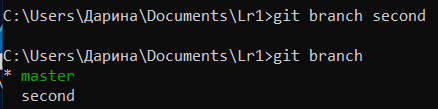


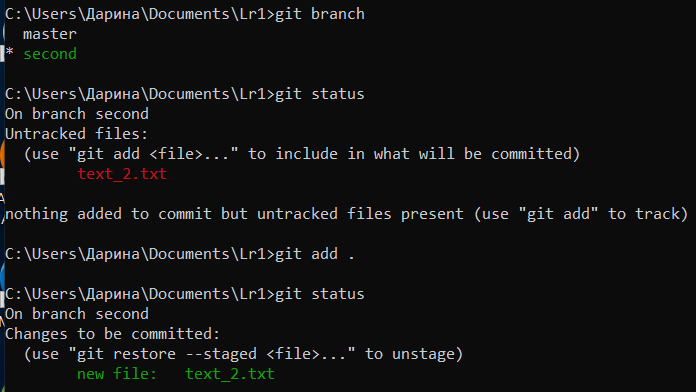
Рисунок 23. Создание второй ветки.

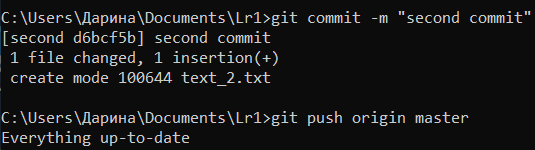
Переключимся на другую ветку, используя команду git checkout second



Рисунок 24. Переключение на вторую ветку.

Создадим новый файл в нашем локальном репозитории и напишем в нем свою фамилию, добавим в область подготовленных файлов, закомитим и отправим на сервер.





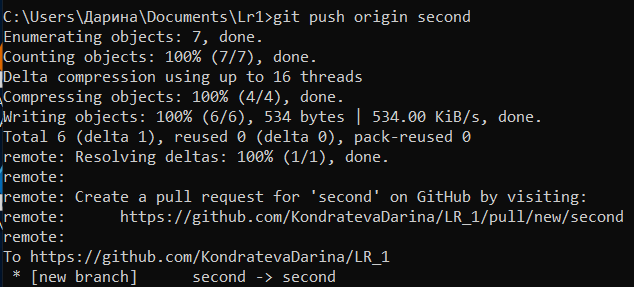


Рисунок 25. Отправка коммита на сервер. Вторая ветка.

Зайдём на github и посмотрим, что в ветке master нет вашего файла, а в ветке second он присутсует.

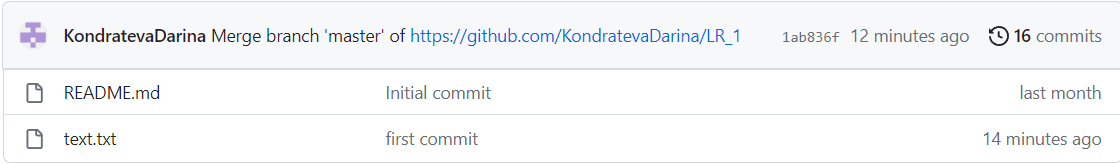


Рисунок 26. Ветка master.

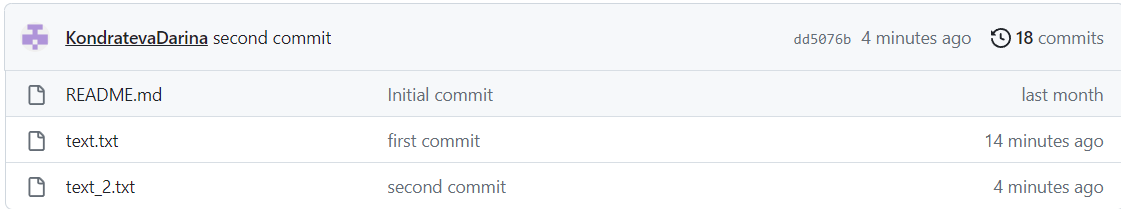


Рисунок 27. Ветка second.

Слияние веток

Переключитесь снова на ветку master. Если через проводник windows в папку с проектом, то файла text\_2.txt не будет виден, потому что мы переключились обратно на ветку master, в которой такого файла не существует. Чтобы он появился, нужно воспользоваться merge для объединения веток (применения изменений из ветки second к основной версии проекта).

git merge second

Если все прошло без ошибок, можно удалить ветку git branch –d second

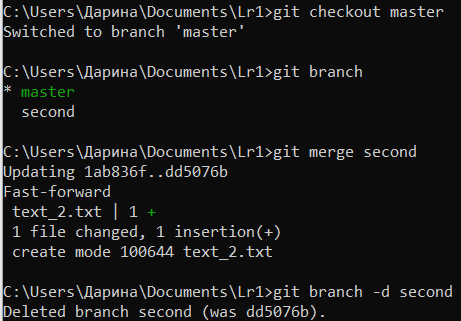


Рисунок 28. Слияние веток и удаление второй ветки.

Создадим ветку с названием newdev

Переключимся на нее

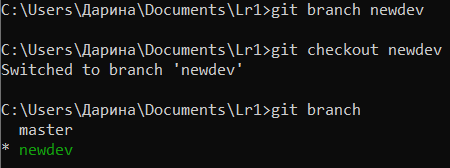


Рисунок 29. Ветка newdev

Добавим в файл с именем отчество

Зафиксируем изменения

Закомитьте изменения

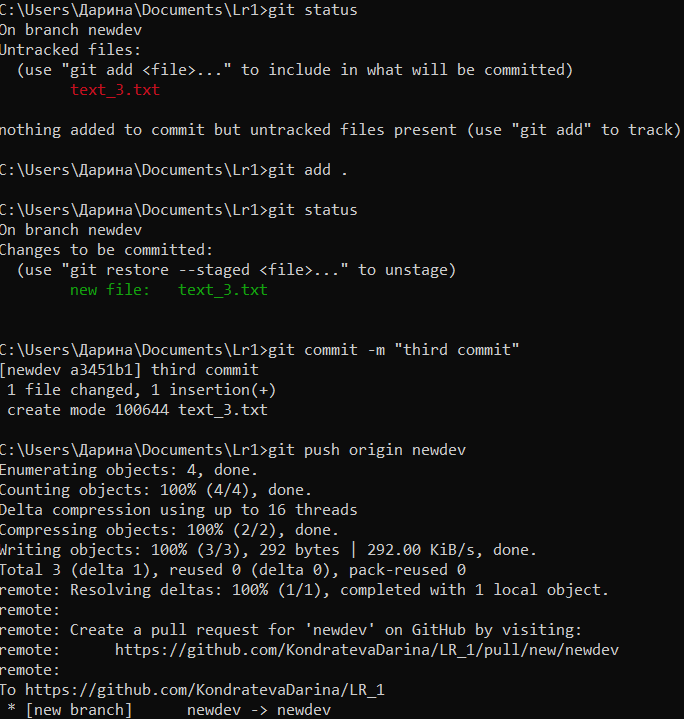
Попробуйте переключиться обратно на ветку master и посмотреть файл: в нем только имя, отчество в другой ветке 

Рисунок 30. Отправка коммита на ветку newdev.

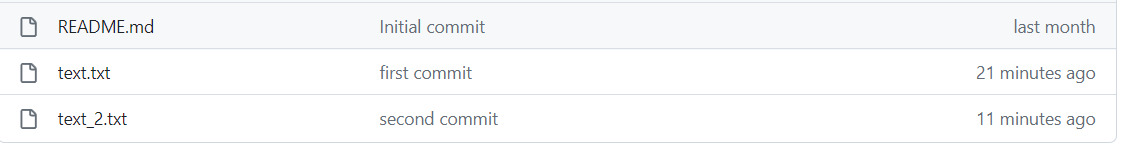


Рисунок 31. Ветка master.

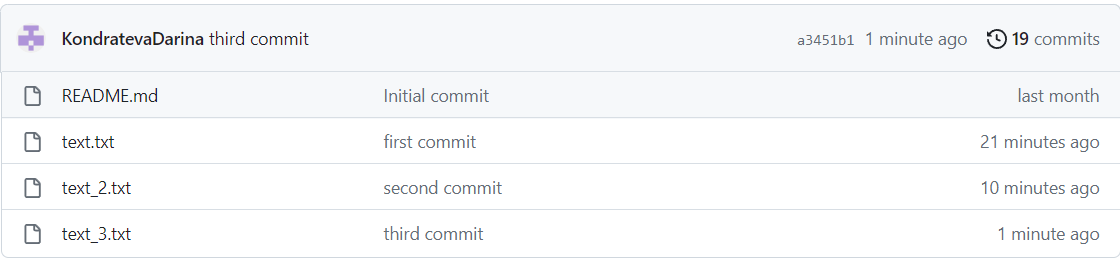


Рисунок 32. Ветка newdev.

Объединим ветки git merge newdev

Откройте файл снова, там есть и имя, и отчество

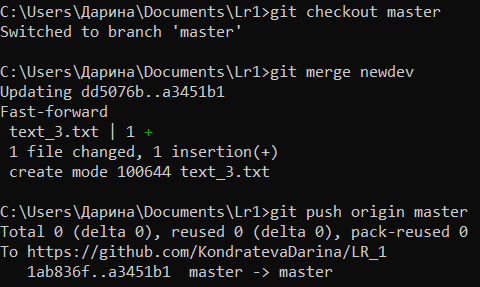


Рисунок 33. Объединение веток.

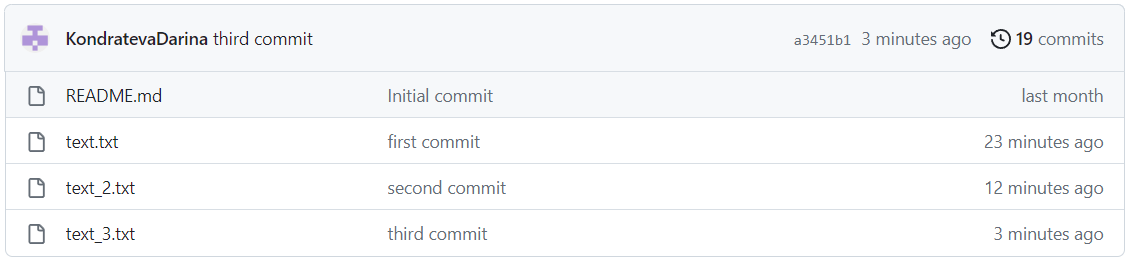


Рисунок 34. Ветка master.

Переключитесь на ветку newdev и удалите в отчестве несколько букв

Зафиксируйте, закоммите

Переключимся на ветку master, добавьте к отчеству несколько букв З

афиксируйте, закоммите

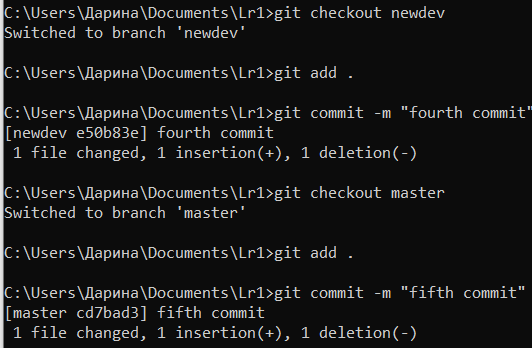


Рисунок 35. Резульатт выполнения команд.

Объединим ветки git merge newdev

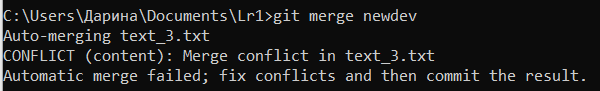


Рисунок 36. Объединение веток.

Теперь ничего не получится, т.к. есть изменения в обеих ветках

Напишем команду, для просмотра изменений

git diff

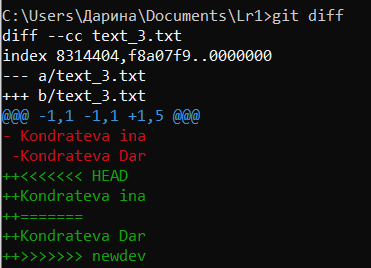


Рисунок 37. Просмотр изменений в файлах.

Иногда конфликты исправляются автоматически, но обычно с этим приходится разбираться вручную — решать, какой код остается, а какой нужно удалить. Система не смогла разрешить конфликт автоматически, значит, это придется сделать разработчикам. Приложение отметило строки, содержащие конфликт:

<<<<<<< HEAD

Евгеньевич

=======

Е

>>>>>>> newdev

Над разделителем ======= мы видим последний (HEAD) коммит, а под ним — конфликтующий. Таким образом, мы можем увидеть, чем они отличаются и решать, какая версия лучше. Или вовсе написать новую. В этой ситуации мы так и поступим, перепишем все, удалив разделители (HEAD, ===, <<>>), и дадим git понять, что закончили. Процесс может быть довольно утомительным и может быть очень сложным в больших проектах. Многие разработчики предпочитают использовать для разрешения конфликтов клиенты с графическим интерфейсом. Выбираем нужную строку (какую строку оставить, может вообще все хотим оставить), удаляем разделители, сохраняем файл, фиксируем изменения, коммитим, отправляем на сервер.

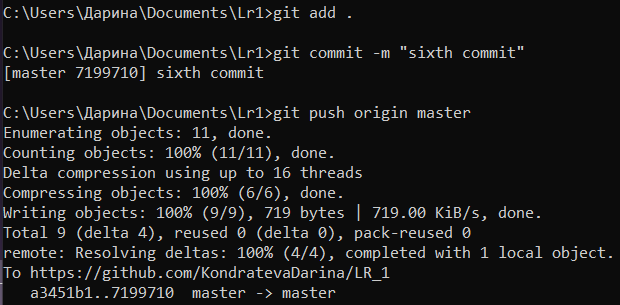


Рисунок 38. Отправка коммита на сервер.

Удалим ветку newdev.



Рисунок 39. Удаление ветки newdev.

Если посмотреть на github, то наши ветки там остались:

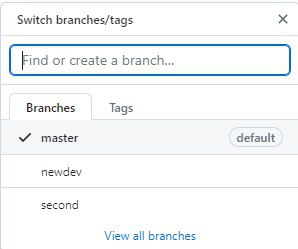


Рисунок 40. Ветки на github.

Давайте удалим их из github

git push origin --delete newdev

git push origin --delete second

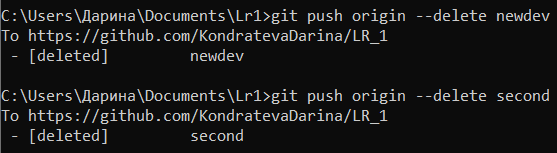


Рисунок 41. Удаление веток на github.

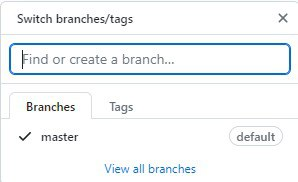


Рисунок 42. Ветки на github.

Возврат к предыдущему состоянию

Гит позволяет вернуть выбранный файл к состоянию на момент определенного коммита. Это делается уже знакомой нам командой checkout, которую мы ранее использовали для переключения между ветками. Но она также может быть использована для переключения между коммитами.

Чтобы посмотреть все комиты, можно использовать команду git log

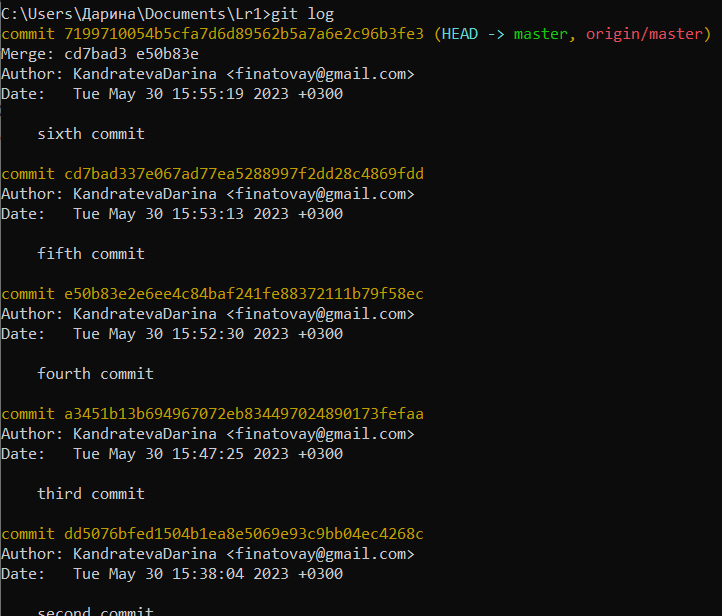


Рисунок 43. git log.

Или можно использовать github или любой клиент

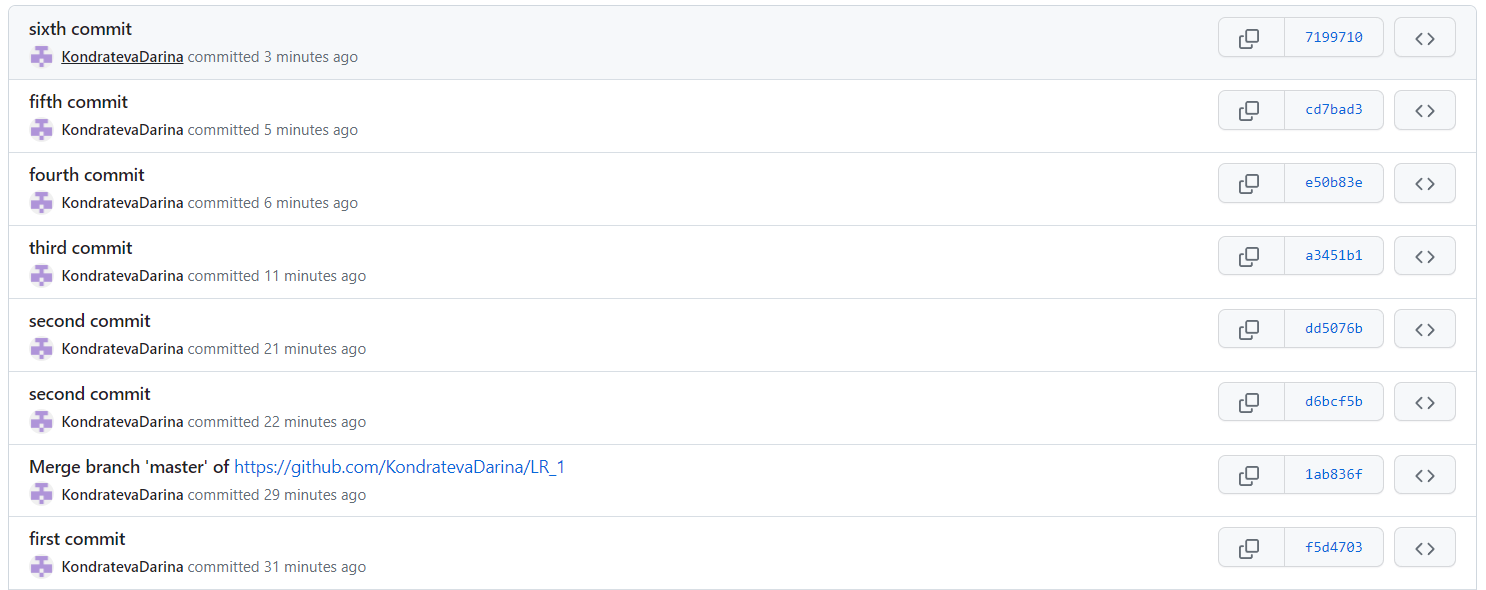
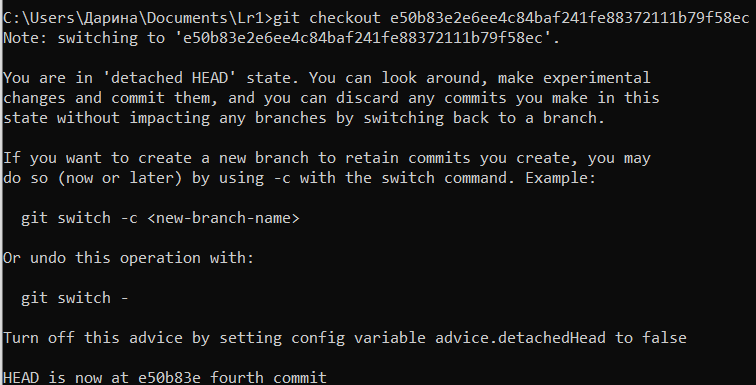


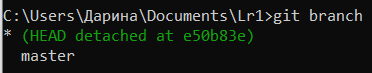
Рисунок 44. Коммиты на github.

Выберем любой коммит, на который хотим откатиться, достаточно указать его первые несколько символов



Если посмотреть файлы теперь, то увидим, что они поменялись – мы вернулись назад.

Чтобы это произошло создалась псевдо-ветка начинающаяся на этом коммите, посмотрите ветки, и мы увидим:

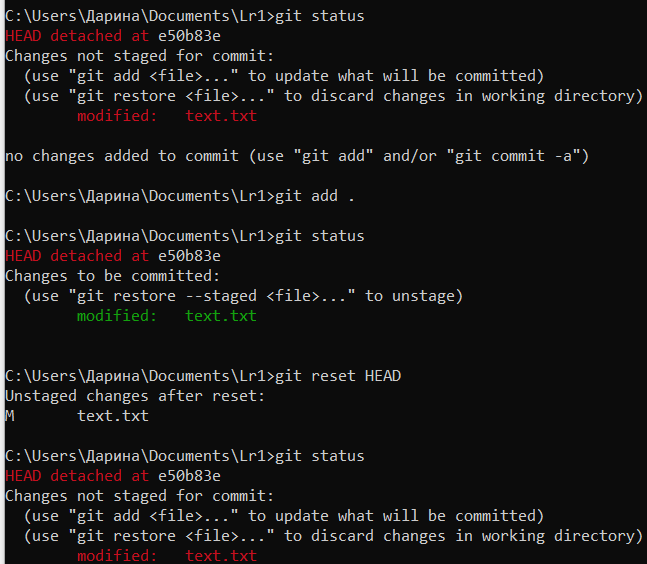


Исправление коммита

Возможна ситуация, что вы закрепили файлы (поменяйте что-то в файле и закрепите их, вызовите git status), но еще не коммитили, хотите убрать файлы из области закрепления, вызовите команду:

git reset HEAD

Ваши файлы останутся такими же, но уйдут из области закрепления (вызовите git status) и снова будет показано, что есть измененные файлы



Отправка только нужных файлов на сервер

В большинстве проектов есть файлы или целые директории, в которые мы не хотим (и, скорее всего, не захотим) коммитить. Мы можем удостовериться, что они случайно не попадут в git add -A при помощи файла «.gitignore»

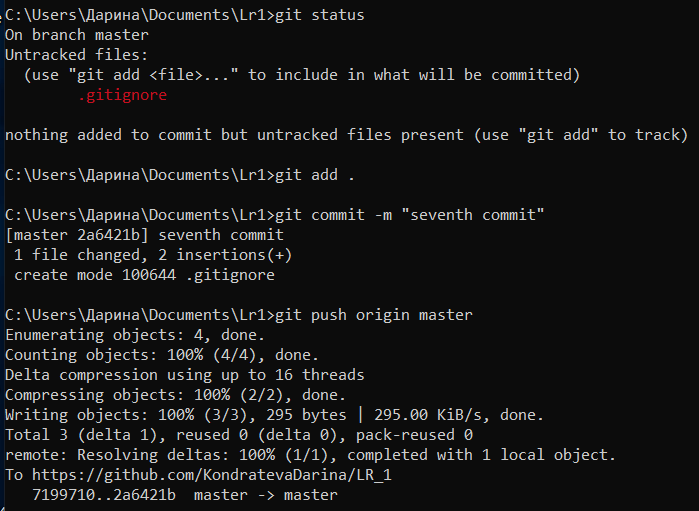
Создайте вручную файл под названием «.gitignore» и сохраните его в директорию проекта.

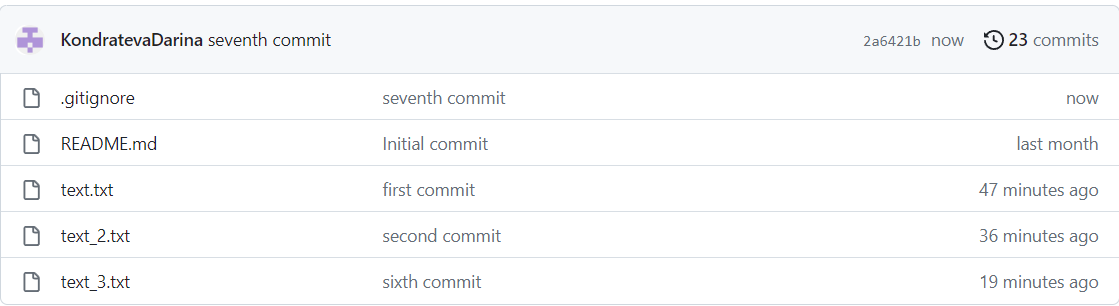
Внутри файла перечислите названия файлов/папок, которые нужно игнорировать, каждый с новой строки.

Файл «.gitignore» должен быть добавлен, закоммичен и отправлен на сервер, как любой другой файл в проекте. Впишите в «.gitignore» файл, который не хотите отправлять на сервер.

Зафиксируйте изменения, закоммите, отправьте на сервер

Все изменения можно проводить непосредственно на github, там также можно делать коммиты, только фиксация изменений происходит при сохранении файла





Совместная работа с git

Выберете себе напарника, скинте друг другу адреса своих репозиториев   
Переходите по полученным ссылкам на репозиторий и нажимаете fork

