ОСНОВЫ GIT

Практическое введение в систему контроля версий

Каждый коммит — шаг к уверенному коду

git

Содержание

Часть 1. Введение в Git

§ 1.1. Что такое контроль версий

Наверное, трудно найти человека, который бы, готовя в школе, техникуме, лицее, институте, университете, на работе (список можно продолжать ещё очень и очень долго) доклад, презентацию или курсовую работу, не создавал под другим именем копию своего файла каждый раз, когда вносил какие-то значительные правки. Возможно, и у Вас где-нибудь есть папка с файлами, которые называются, например, так:

- Презентация.pptx
- Презентация1.pptx
- Презентация2.pptx
- Презентация черновик.pptx
- Презентация показать руководителю.pptx
- Презентация последняя версия.pptx
- Презентация последняя версия точно.pptx
- Презентация последняя версия точно НЕ МЕНЯТЬ.pptx
- Презентация последняя версия точно НЕ МЕНЯТЬ 22 ноября.pptx

и так далее, и так далее. Если вам это знакомо, то вы уже сталкивались с концепцией контроля версий на самом интуитивном уровне. Но нет ли способа решить эту задачу проще и, главное, лучше?

Конечно, есть. Программные решения, берущие на себя эту заботу, называются **системами контроля версий**. На текущий момент есть много разных систем контроля версий: Git, Subversion (известная также как SVN), Mercurial, Visual Source Safe (о котором сегодня вспоминают как о страшном сне из прошлого тысячелетия – причины такого отношения станут понятны в следующем параграфе) и другие. Использование системы контроля версий – это общий стандарт в промышленной разработке программного обеспечения. А вот какую именно систему взять – это в каждой организации (а иногда и в каждой рабочей группе) устанавливается отдельно. Каждая из названных систем решает задачу отслеживания изменений в вашем проекте и управления ими. Однако, «золотой стандарт» на момент написания этого учебника – это Git, поэтому о нём далее речь и пойдёт.

Система контроля версий состоит из различных частей. Основные из них – это ядро системы, облачный хостинг (в принципе не строго обязателен, но необходим при многопользовательской разработке или даже при однопользовательской разработке, но с нескольких машин... или даже с одной машины, но если мы хотим

иметь резервные копии в облаке) и клиентское программное обеспечение. Даже если все пользуются одним ядром (например Git), у каждого из вас может стоять самое разное клиентское ПО, которое позволяет с этим самым Git работать: Atlassian SourceTree, Tortoise Git, Git Bash, ну и просто встроенные в Visual Studio или в Visual Studio Code инструменты взаимодействия с ядром Git. Какой из них использовать – это уж как вам удобнее (например, лично я использую встроенные инструменты Visual Studio и Visual Studio Code для рядовых операций, Tortoise Git для не вполне тривиальных действий и командную строку Git Bash для сложных случаев, когда проще вспомнить команду, чем искать, где её реализация в графическом интерфейсе, если она вообще есть).

Git предлагает множество мощнейших команд для работы с историей изменения проекта. Мы постепенно изучим их от простых и наиболее употребимых к сложным и редко используемым (но тем не менее иногда незаменимым). А сейчас важно усвоить следующее:

1. Git позволяет сохранять текущее состояние проекта (и не одного файла, а всех входящих в проект файлов) в разные моменты времени. Каждый такой «снимок» состояния проекта называется **коммитом** или **фиксацией**.

2. Мы всегда можем сравнить между собой два коммита, а также сравнить любой коммит с текущим состоянием проекта, и таким образом увидеть, какие файлы изменились, и что именно изменилось.

3. Если какое-то изменение «сломало» проект, мы всегда можем откатиться к тому состоянию проекта, которое было до него (а если это изменение было не последним, то сделать «обратное» изменение).

4. При многопользовательской работе над проектом мы всегда можем посмотреть, кто именно сделал то или иное изменение и когда.

5. Мы можем создавать **ветки** – одновременно существующие версии одного и того же проекта – и переключаться между ними. Можно считать, что ветки – это разные пути развития проекта, а коммиты – это остановки на этих путях. Ветки позволяют, например, работать над новой функцией вашего продукта изолированно, не нарушая целостность рабочего кода и, таким образом, не теряя возможности править ошибки, обнаруженные в текущей версии этого продукта. Кроме того, разные разработчики могут в одно и то же время работать над разными функциями в разных ветках, не мешая друг другу, а в нужный момент эти функции сливаются в основную ветку (часто после проверки разными участниками команды), причём таким образом, чтобы сделанные изменения не нарушали работу друг друга и остальных частей продукта.

Все эти возможности делают системы контроля версий незаменимой частью процесса разработки программного обеспечения.

Платформа облачного хостинга, такая, как GitHub или GitLab, также может предоставлять дополнительные возможности работы над проектом – например, осуществлять непрерывную поставку новых версий программного обеспечения конечному пользователю.

В любом случае, используя систему контроля версий, будь то Git или любая другая, вы можете навсегда забыть об аде почти одинаковых файлов с многочисленными «ПОСЛЕДНЯЯ ВЕРСИЯ» и «УЖ ТОЧНО ПОСЛЕДНЯЯ ВЕРСИЯ». В рабочей папке у вас всегда находится одна и только одна версия каждого файла, а к каждой другой версии всегда есть достаточно подробное (впрочем, это уже от вас зависит) описание и (а вот это Git предоставит сам) чёткая карта того, чем эта версия отличается от предыдущей.

Вопросы для самопроверки:

1. Что такое коммит?

   Варианты ответов:

   1.1. То же, что и ветка разработки.

   1.2. Снимок состояния папок и файлов проекта.

   1.3. Облачный сервис для резервного копирования.

   1.4. Инструментарий для совместной разработки.

2. Для чего служат ветки?

   Варианты ответов:

   2.1. Они позволяют автоматически исправлять ошибки в проекте.

   2.2. Они позволяют ускорить работу вашего приложения.

   2.3. Они позволяют работать над новыми функциями изолированно от рабочего кода.

   2.4. Они позволяют защитить ваш код с применением современных методов криптографии.

3. Что из перечисленного НЕ является системой контроля версий?

   Варианты ответов:

   3.1. Git

   3.2. Subversion

3.3. Mercurial

3.4. Docker

§ 1.2. Немного об истории систем контроля версий

Путь систем контроля версий к нынешнему состоянию был долог и многоэтапен.

К идее того, что ад множественных копий одного документа – это плохо, пришли довольно быстро, так что первые системы контроля версий появились едва ли не раньше, чем персональные компьютеры. Однако они не только внешне (что естественно), но и по способу своей работы существенно отличались от того инструментария, которым мы пользуемся сейчас.

Первое поколение: эра пессимистических блокировок

Возможно, первой в истории сколько-нибудь успешно внедрённой системой контроля версий была разработанная в 1972 году SCCS – Source Code Control System. Однако, рискну предположить, что для большинства российских разработчиков программного обеспечения первой такой системой стал Visual SourceSafe, или VSS (изначально разработан компанией One Tree Software, в 1994 году компания поглощена корпорацией Microsoft). Поскольку обе системы работают в общем и целом на одинаковых принципах, позволю себе объединить их в одно поколение, хотя их разделяет двадцать с лишним лет и хотя на момент появления Visual SourceSafe уже начали появляться системы контроля версий, реализующие более продвинутые модели использования.

Как были устроены SCCS и VSS? Репозиториев как таковых не было. Были доступные по локальной сети папки проекта на файл-сервере (или просто на какой-то выделенной машине). Все файлы проекта хранились с атрибутом «только для чтения». Если разработчик хотел просто получить свежую версию кода, он выполнял команду get (для SCCS) или выбирал эту команду из меню (для VSS). При этом он получал себе последнюю версию файла в режиме «только для чтения». Если нужно было внести изменения в какой-то файл, нужна была другая команда – edit для SCCS или check out для VSS. В таком случае разработчик получал файл в режиме доступа на чтение и запись, но при этом система контроля версий следила, чтобы больше никто этот файл в таком режиме получить не мог. После того, как разработчик закончил работу, он должен был записать изменения на файл-сервер командой delta для SCCS или check in для VSS. При этом файл переходил обратно в режим доступа только на чтение, а блокировка с файла снималась, т.е., другой разработчик мог получить доступ на запись.

Таким образом, каков бы ни был размер команды разработчиков, никакие два сотрудника не могли одновременно работать с одним и тем же файлом. Если же это было необходимо, один сотрудник вынужденно тормозил работу другого (или других). Отсюда правило, которое существовало в те времена в большинстве компаний, разрабатывающих программное обеспечение – кусок работы, производимой за один раз, не должен быть больше, чем можно успеть сделать до

конца рабочего дня, и перед уходом с работы каждый сотрудник обязан «отпустить» все «занятые» им файлы.

Такая модель работы с файлами называется моделью пессимистических блокировок. Почему пессимистических? Потому что такая блокировка как бы неявно предполагает, что одновременная работа нескольких сотрудников с одним файлом обязательно и неминуемо приведёт к тому, что изменения, производимые одним сотрудником, перезапишут изменения, производимые другим сотрудником (собственно, учитывая, что вместо сервиса репозитория была просто общая папка в локальной сети, так оно и было), а значит, её нельзя допускать, и открытие файла на запись одним сотрудником должно запрещать редактирование файла всеми остальными.

Торможение совместной работы, вызванное необходимостью пессимистической блокировки файлов, было одним из минусов такой организации контроля версий. Другой минус – это то, что изменения каждого файла записывались фактически независимо от изменения других файлов. Это, на самом деле, также было неминуемым следствием пессимистической блокировки. Допустим, два сотрудника работают над одним проектом, и в этом проекте есть файл, который время от времени приходится дорабатывать обоим. В таком случае, пока они одновременно работают с разными файлами, они друг другу не мешают, но начинают мешать, как только пишут что-то в этот общий файл. Поэтому, чтобы сократить время простоя друг друга, каждый раз при внесении новой функции в общий файл сотрудник открывает этот файл на запись, вносит изменения и тут же «отдаёт» файл. Таким образом, хотя один конкретный сотрудник работает над одной задачей (например, «реализовать набор функций для расчёта и рисования гистограмм»), он многократно сохраняет состояние одного и того же файла. Другой сотрудник в то же время также многократно сохраняет состояние того же файла, работая над другой задачей (скажем, «реализовать набор функций для расчёта и рисования сплайн-интерполяции»). Потом по истории изменений этого файла сложно будет объединить все изменения, сделанные в рамках работы над одной конкретной задачей, и невозможно в случае необходимости отменить их одним движением.

Таким образом, хотя уже и это было лучше, чем вообще ничего – всё-таки появилась история изменений каждого конкретного файла, пропала необходимость держать лютый ад множественных копий одного файла с небольшими отличиями и как-то в нём разбираться – системы контроля версий первого поколения были ещё далеки от идеала.

Второе поколение: оптимистические блокировки и атомарные коммиты

История второго поколения систем контроля версий начинается в середине 80-х, т.е., спустя полтора десятилетия после появления первого поколения, с появления Concurrent Versions System, или CVS. Однако, опять же, рискну предположить, что в России лавинообразный переход на второе поколение состоялся в начале 2000-х, с появлением системы Subversion, или SVN, в которой

функции, характерные для второго поколения, были реализованы значительно лучше и надёжнее, нежели в CVS.

Наиболее значительным шагом вперёд в системах второго поколения был переход от хранения проекта в общей сетевой папке к полноценному сетевому сервису репозитория, что, в свою очередь, позволило перейти от пессимистической модели блокировок к оптимистической, а это, далее, позволило перейти от записи изменений каждого файла отдельным действием к системе коммитов – наборов изменений, объединённых общей целью.

Системы второго поколения опираются на централизованный репозиторий, к которому каждый разработчик подключается при выполнении действий с системой. Пессимистических блокировок больше нет – все файлы доступны на запись в любой момент. Разработчик может вносить изменения в проект, не оглядываясь на коллег, а когда нужно, сохранять эти изменения одной командой. В случае, если другие разработчики не вносили изменения в те же файлы, эти изменения в принципе друг другу не мешают. Если несколько разработчиков вносили изменения в один и тот же файл (как в примере про двух разработчиков и общий файл), но при этом не правили один и тот же кусок файла, в итоговый файл записываются оба изменения. Сложность наступает только если два пользователя внесли изменения в одно и то же место одного и того же файла – такая ситуация называется конфликтом, и тогда тот, кто не успел первым, должен самостоятельно исправить конфликт – расставить изменения в нужном порядке, если нужны оба, или оставить в итоговом файле те изменения, которые нужны, убрав те, которые при этом излишни. Такая модель называется оптимистической блокировкой. Оптимистической – потому что предполагается, что в большинстве случаев конфликтов не возникает.

В Subversion, возможно, впервые реализованы атомарные коммиты – по окончании работы над задачей разработчик делает коммит, и изменения во всех затронутых файлах либо все записываются, либо, при возникновении нештатной ситуации, все не записываются. Частичной записи, как в CVS, когда при возникновении ошибки репозиторий мог остаться в несогласованном состоянии, в SVN не будет. Это позволяет свести воедино изменения, осуществлённые в рамках работы над конкретной задачей, и, при необходимости, пакетно их отменить.

Для оптимизации по занимаемому дисковому пространству первая версия каждого файла в Subversion (если ориентироваться на него), записывается целиком, а при последующих коммитах записываются только внесённые изменения. Однако это приводит к тому, что, если изменений много, получение текущей версии может занять существенное время (нужно ведь получить первую версию, получить все внесённые изменения и последовательно их применить). Поэтому в Subversion, на самом деле, реализован компромисс – на каждый файл сохраняется до 1023 изменений подряд, а дальше опять делается полная копия текущей версии файла.

В Subversion также реализована возможность создания веток репозитория, пока ещё в виде отдельных папок. В репозитории создаются три папки: trunk, branches и tags. В папке trunk хранится основная ветка кода, из которой осуществляется поставка стабильных версий разрабатываемого программного обеспечения. Для каждой ветки создаётся подпапка в папке branches. Наконец, в папке tags хранятся теги – версии кода, помеченные отдельными метками (обычно теги ставят для маркировки релизов).

Второе поколение систем контроля версий стало существенным скачком вперёд – разработчики больше не тормозили работу друг друга пессимистическими блокировками, а история изменений стала соответствовать работе над конкретными задачами. Однако простор для улучшений ещё оставался. Например, что будет, если связь с централизованным репозиторием пропадёт?

Третье поколение: распределённые системы

Дальнейшим развитием стало третье поколение систем контроля версий, к которому и относится Git. В отличие от Subversion, где центральный репозиторий был один, и коммит производился туда, в Git используется модель распределённого хранения. На машине каждого сотрудника имеется полностью функциональный репозиторий, и каждый разработчик выполняет коммиты именно в свой локальный репозиторий на своей машине. Коммиты, уже выполненные локально, можно отправить на удалённый репозиторий, а также можно получить с удалённого репозитория коммиты, отправленные туда другими разработчиками. Удалённый репозиторий, для организации командной работы, таким образом, тоже существует, но пропажа связи с ним не затормаживает работу – максимум, на какое-то время сотрудник лишается обновлений, сделанных коллегами. Кроме того, даже ликвидация удалённого репозитория (например, блокировка связи с ним) не станет помехой – можно создать удалённый репозиторий в другом месте, склонировать туда локальный репозиторий одного из сотрудников и отправить в него изменения из остальных локальных репозиториев. Более того, можно даже иметь несколько удалённых репозиториев одновременно.

Другое изменение в Git по сравнению с SVN – для каждого коммита хранится не список изменений (дельты) для каждого файла, а «снимки» состояния файловой системы (при этом файлы, которые с предыдущей версии не изменились, Git не дублирует). Есть также и другие нововведения – например, промежуточная стадия индексированных файлов – мы с ними столкнёмся по мере изучения Git.

Вопросы для самопроверки:

1. Как называется принцип работы, когда один файл в одно время может редактировать только один разработчик?

   Варианты ответов:

1.1. Модель пессимистических блокировок.

1.2. Модель оптимистических блокировок.

1.3. Модель блокирующего редактирования.

2. Что такое атомарный коммит?

   Варианты ответов:

   2.1. Коммит, который содержит изменения, относящиеся к решению одной конкретной задачи.

   2.2. Коммит, который содержит изменения только одного файла.

   2.3. Коммит, который не содержит добавления новых или удаления старых файлов.

3. В чём состоит главное отличие Git от Subversion?

   Варианты ответов:

   3.1. Subversion использует модель пессимистических блокировок, а Git оптимистических.

   3.2. В Subversion коммиты хранятся на сервере, а Git строго локальная система.

   3.3. Git является распределённой системой, которая сохраняет работоспособность при потере сервера, а Subversion привязана к серверу.

§ 1.3. Что такое GitHub и GitLab

Ранее мы упомянули GitHub и GitLab. Это, как мы уже сказали, программное обеспечение для хостинга систем контроля версий. Это означает, что GitHub и GitLab – облачные решения, которые позволяют хранить ваши проекты, находящиеся под контролем версий (или, говоря короче, **репозитории**) и позволяют работать над ними нескольким людям и/или с нескольких машин. Поэтому GitHub и GitLab называют облачными репозиториями или веб-репозиториями. Важно понимать, что Git – это распределённая система, и она отлично работает локально на компьютере пользователя и без каких-либо серверов. Однако, наличие облачного хостинга позволяет не только хранить резервные копии своих репозиториев, но и обмениваться их содержимым между участниками проекта.

Многие из проектов, которые хранятся на GitHub или GitLab, представляют собой проекты с открытым исходным кодом, или open-source. Это означает, что все пользователи того облачного репозитория, на котором хранится такой проект, могут видеть публикуемый в нём код, предлагать свои изменения, сообщать о проблемах и даже создавать форки (от английского «fork» – «вилка») – свои проекты, основанные на существующих проектах с открытым исходным кодом.

Но не всегда уместно создавать проект с открытым исходным кодом. Зачастую бывает так, что нужно предоставить возможность видеть код и работать над ним только некоторым людям, составляющим команду разработчиков. В таком случае вы можете опубликовать свой проект как проект с закрытым исходным кодом. Тогда видеть код проекта и работать над ним смогут только те люди, которым вы явным образом дали доступ.

Вне зависимости от того, работаете вы над проектом с открытым или закрытым исходным кодом, облачные репозитории значительно облегчают жизнь команде разработки.

Например, предположим, что кто-то из разработчиков зашёл в тупик и хочет спросить совета у ведущего разработчика. Вместо того, чтобы пересылать куски кода или весь проект через мессенджер или устраивать созвон с демонстрацией экрана, он может выполнить коммит кода, с которым имеет проблемы, в отдельную ветку репозитория и отправить эту ветку в облако. После этого всем членам команды этот код доступен, и ведущий разработчик может посмотреть его, запустить у себя, внести изменения (причём можно это делать даже в браузере на самом сайте облачного репозитория – впрочем, лучше этой возможностью пользоваться только для небольших изменений, какие-то крупные изменения значительно удобнее делать локально в соответствующей среде разработки на компьютере), и всегда можно будет увидеть, какие именно изменения были произведены.

Теперь допустим, что кто-то из разработчиков завершил создание новой функции проекта. Он может отправить ветку с этой функцией в облако и создать запрос на внедрение новой функции в проект (на GitHub это называется Pull Request или PR, а в GitLab – Merge Request или MR). Ведущий разработчик может проанализировать работу коллеги, при необходимости оставить к ним комментарии или внести правки, а когда новая функция действительно, по его мнению, готова к внедрению, аккуратно внедрить её в рабочий код (эта операция называется Merge или Слияние). После этого все участники команды могут получить на свои рабочие места актуальную версию кода и учесть сделанные изменения, когда будут работать над следующей задачей.

Кстати, раз уж зашла речь о задачах, многие облачные репозитории предоставляют также и **систему управления задачами**. Ведущий разработчик может использовать их для раздачи заданий членам команды, отслеживать их выполнение, принимать от пользователей сообщения об ошибках и даже планировать **спринты** (короткие циклы разработки), если команда использует их в организации своей работы.

Также облачные репозитории помогают организовывать конвейеры непрерывной поставки кода, когда каждое влитое ведущим разработчиком в рабочий код изменение вызывает, например, построение новой версии приложения, запуск системы контроля качества кода, прогон тестов и поставку новой версии на демонстрационный стенд.

Программное обеспечение некоторых облачных репозиториев – например, GitLab – также поставляется на условиях открытой лицензии. То есть, если компания по каким-либо причинам не хочет доверить хранение своего репозитория самому GitLab, она может создать свой облачный репозиторий с точно таким же программным обеспечением в своей корпоративной сети. Кроме того, если говорить о программном обеспечении GitLab, оно также поставляется с открытым исходным кодом, так что желающие могут создать свой форк, доработать по-своему и создать свой облачный репозиторий уже с изменённым под себя программным обеспечением. Скажем, некоторые компании предпочитают не полагаться на внешние сервисы и разворачивают GitLab на своих серверах. Существуют и целые платформы хостинга репозиториев на основе форков GitLab – например, школьникам для участия в олимпиадах по информатике предписано пользоваться решением МосХаб, в котором система авторизации интегрирована с единым российским поставщиком авторизации ЕСИА (единой системой идентификации и аутентификации). Самостоятельное развёртывание собственного сервера GitLab – отдельная и большая тема, которой обычно занимаются специалисты технической поддержки, системные администраторы или инженеры DevOps. Однако работа пользователя с таким сервером не отличается от работы с публичным хостингом Git – просто нужно знать URL хостинга и используемый способ аутентификации.

Вопросы для самопроверки:

1. Какова основная цель таких решений, как GitHub и GitLab?

   Варианты ответов:

   1.1. Хранение резервной копии кода.

   1.2. Предоставление инструментария для командной разработки.

   1.3. Эти платформы предназначены исключительно для проектов open-source.

   1.4. Инструментарий для редактирования кода онлайн.

2. В чём польза GitHub и GitLab для командной разработки?

   Варианты ответов:

   2.1. Они автоматически исправляют ошибки в коде.

   2.2. Они избавляют от необходимости анализа кода ведущим разработчиком.

   2.3. Они позволяют членам команды видеть сделанные друг другом изменения и совместно работать над ними.

   2.4. Они обеспечивают бесплатный хостинг разрабатываемых веб-приложений.

3. В чём польза GitHub и GitLab для руководителя проекта?

   Варианты ответов:

   3.1. Автоматическая генерация шаблонного кода.

   3.2. Наличие встроенной системы управления задачами.

   3.3. Интеграция с современными мессенджерами.

   3.4. Автоматическая регистрация доменного имени для веб-приложения.

§ 1.4. Установка и настройка Git

Для того, чтобы начать пользоваться Git, прежде всего, разумеется, надо его установить и настроить.

Для начала проверим, есть ли он у вас. Зайдите в терминал или оболочку командной строки, в зависимости от вашей операционной системы, и запустите следующую команду:

```
git --version
```

Если Git установлен, вы увидите сообщение с номером версии:

```
git version 2.45.1.windows.1
```

Если вы видите не сообщение от Git с номером версии, а что-то другое (сообщение о том, что нет такой команды, например), значит, Git, скорее всего, не установлен, а значит, нам надо его установить (возможно) и настроить (определённо).

Если ваш компьютер работает под управлением Linux, то, скорее всего, Git уже установлен (но, возможно, не применены начальные настройки). Если вдруг это не так, вам следует установить пакет «git» из менеджера пакетов вашей сборки Linux. Так, для сборок Linux на основе Debian вам понадобится выполнить команды:

```
sudo apt-get update

sudo apt-get install git
```

Для похожей, но всё же другой сборки может потребоваться сначала перейти в режим суперпользователя. Например, в Alt Linux:

```
su -
```

а затем уже установить Git:

```
apt-get update

apt-get install git
```

А для сборок Arch Linux потребуется обратиться к другому менеджеру пакетов:

```
sudo pacman -S git
```

Если у вас компьютер под управлением macOS, вы можете установить Git через популярный менеджер пакетов Homebrew (если он у вас уже установлен):

```
brew install git
```

или скачать установщик с сайта Git (git-scm.com).

Если у вас компьютер под управлением Windows, вы можете либо скачать установщик с сайта Git (git-scm.com), либо, если вы пользуетесь менеджером пакетов Chocolatey, зайти в терминал PowerShell и выполнить команду:

```
choco install git.install
```

При установке Git под Windows он будет задавать множество уточняющих вопросов относительно состава вносимых изменений и настроек. Поскольку они нам сейчас ничего не скажут, а также поскольку все их можно перенастроить позднее, когда у вас будут собственные предпочтения, лучше оставить всё по умолчанию. Особое внимание стоит уделить, пожалуй, вопросу о том, как поступать с переменной PATH – системной переменной, которая позволяет вызывать из командной строки приложение без указания полного пути к нему. По умолчанию обычно предлагается согласиться с внесением изменений в PATH, но если вдруг это не так – надо всё же согласиться.

После того, как установка завершилась, проверьте, что Git действительно установился, выполнив те же действия, что и в начале параграфа.

Для внесения изменений в настройки Git используется команда `git config` с разными ключами и параметрами (config – это сокращение от английского configuration – конфигурация, т.е. настройки). Некоторые из настроек нам после установки Git в любом случае нужно установить.

Настройки Git делятся на три уровня. Системный уровень – это настройки, действие которых распространяется на все операции с Git на данной машине. Под Linux они, как правило, хранятся в файле /etc/gitconfig, под Windows – в файле gitconfig в подпапке etc папки, куда установлен Git. Глобальный, или пользовательский, уровень – это настройки, действие которых распространяется на все операции с Git со всеми репозиториями на данной машине, но только для конкретного пользователя. Для другого пользователя эти настройки могут быть другими. Эти настройки, как правило, хранятся в файле .gitconfig в домашней папке пользователя. Наконец, локальный уровень настроек – это настройки, действие которых распространяется на данный конкретный репозиторий на данной машине. Такие настройки обычно хранятся в файле .gitconfig в самом репозитории.

Команда git config без ключей и параметров просто выводит перечень своих ключей и параметров с кратким описанием, какой из них для чего служит.

Попробуем посмотреть, какие настройки Git уже есть на вашей машине. Если Git только что установлен, то, вероятно, никакие (ну или только те, которые были заданы в процессе установки). Выполним команду

```
git config --list
```

Эта команда выводит все сделанные настройки и их значения без указания уровня настроек (слово list – это английское слово, означающее список). Чтобы

включить в выдачу уровень настройки (в виде файла, из которого взята та или иная настройка), используйте флаг --show-origin:

```
git config --list --show-origin
```

По-английски «show origin» означает «показать происхождение».

Git хранит информацию об авторах коммитов. Поэтому для того, чтобы иметь возможность делать коммиты, нам необходимо указать в настройках своё имя (настройка user.name) и адрес электронной почты (настройка user.email). Правильность их, конечно же, никто проверять не будет (по крайней мере пока вы не работаете в крупной компании), но лучше, разумеется, ввести достоверные данные. Например, так:

```
git config --global user.name "Вася Пупкин"

git config --global user.email "basilio.poop@yandex.ru"
```

Эти данные будут «вшиваться» в каждый сделанный вами коммит. Если адрес электронной почты указан неправильно, то коллеги не смогут связаться с вами через систему (например, через упоминания пользователя в комментарии).

А что за флаг `--global`? Как вы, вероятно, догадались, это указание на то, что мы производим настройку на глобальном уровне – то есть, на уровне текущего пользователя. Делать такую настройку на системном уровне неправомочно – если на компьютере есть или когда-нибудь будет учётная запись другого человека, который также будет пользоваться Git, логично, что у него эти две настройки будут другими. А на локальном уровне её делать нелогично – ведь тогда придётся повторять её для каждого репозитория (впрочем, если у нас есть репозиторий, для которого действительно мы должны представиться по-другому – например, одни репозитории содержат корпоративные проекты, и там e-mail нужно указывать корпоративный, а другие свои частные, и там нужно указывать частный), то, конечно, можно для конкретного репозитория указать другое значение на локальном уровне. Если для какого-то репозитория существует какая-то локальная настройка при том, что тот же параметр настроен на глобальном и/или системном уровне, Git для данного проекта возьмёт локальную настройку. Если какой-то параметр задан на глобальном для определённого пользователя и системном уровне, то для данного пользователя Git будет брать глобальную (пользовательскую) настройку.

А почему имя параметра указано как два слова, разделённые точкой? Потому что параметры для удобства сгруппированы по области их применения. В данном случае user – это имя группы параметров, а name и email – имена параметров внутри группы. Поэтому полное имя (а в командах указывается всегда полное имя параметра, чтобы не перепутать параметры с одинаковым именем из разных групп) будет user.name и user.email соответственно.

В определённых случаях Git при работе с командной строки требует, чтобы пользователь что-то ввёл или отредактировал. С этой целью он запускает тот или иной редактор. По умолчанию он использует входящий в состав большинства сборок Linux редактор Vim (для того, чтобы использовать Git с командной строки под Windows, настоятельно рекомендуется использовать входящую в комплект Git программу Git Bash – интерпретатор команд командной строки Linux в Windows). Это довольно мощный (для редактора в текстовом – консольном – режиме) редактор, имеющий много нужных для разработчика или для системного администратора функций, но для работы с ним требуется знать его команды (без этого не получится с первой попытки даже выйти из этого редактора, да-да!). Самый краткий базовый минимум для работы с Vim, которого будет достаточно для взаимодействия с Git, будет приведён здесь немного позже, а если хотите его освоить в достаточной для разработчика или системного администратора степени, рекомендую входящую в комплект Vim (а значит, и Git) программу vimtutor. Эта программа создаёт копию файла с инструкциями и открывает его в самом Vim, после чего команды редактора можно осваивать по мере чтения файла. Если с английским у вас не очень хорошо, я в своё время выполнил перевод этого файла на русский язык и прилагаю его к данному учебнику отдельным файлом. Получить его можно по ссылке:

https://urbrato.ru/textbooks/vimtutor.ru.txt

Обратите внимание – в браузере файл откроется, скорее всего, в нечитаемом виде. Следует сохранить его в какую-нибудь папку. Далее, если вы работаете в Windows, то в этой папке запустить Git Bash. Если у вас нет команды запуска Git Bash в отдельной папке, то запустить Git Bash как получится, а в нужную папку перейти уже в Git Bash при помощи команды

```
cd путь к вашей папке
```

Следует учесть, что в Linux (и в Git Bash) запись пути к папке отличается от принятой в Windows: косые чёрточки развёрнуты в другую сторону, а имя диска указывается не как буква и двоеточие, а как косая черта и буква – например, не D:\Data\file.txt, а /d/Data/file.txt. Не нужно этого бояться. Вместе с тем, вариант «как в Linux» будет по крайней мере гарантированно работать на любой машине (под управлением Linux или macOS это так изначально, а под Windows так при использовании Git Bash).

Оказавшись в нужной папке, выполните в Git Bash команду

```
vi vimtutor.ru.txt
```

Откроется текст, который можно читать и параллельно выполнять указанные в нём действия, так же, как и при выполнении команды vimtutor, только по-русски.

Редактор Vim действительно весьма удобен для редактирования текстовых файлов – настроечных, служебных и т.п. Однако, если хотите, то можно настроить Git на работу с другими редакторами – чаще всего используются Nano (редактор под Linux с более дружественным пользователю интерфейсом), Emacs, Notepad++

(под Windows) или VS Code (с флагом --wait, чтобы управление не вернулось к Git, пока пользователь не выполнит редактирование в VS Code). Для задания редактора служит настройка `core.editor`. Например:

```
   git config --global core.editor emacs

   git config --global core.editor nano

   git config --global core.editor
"C:\Program Files\Notepad++\notepad++.exe"

   git config --global core.editor "code --wait"
```

Однако, я настоятельно рекомендую освоить хотя бы базовый минимум Vim – во-первых, это полезный опыт, а во-вторых, со временем вы оцените его удобство.

Заметим, что в некоторых случаях мы указали редактор без кавычек, а в некоторых случаях – в кавычках. Это связано с тем, что в случаях, когда в состав команды для вызова редактора входит пробел, мы должны заключать команду в кавычки, иначе пробел воспринимается как разделитель параметров команды git config.

После того, как вы настроили имя пользователя и e-mail (а может быть, и редактор), посмотрите ещё раз на список настроек. Вы увидите среди них и только что сделанные настройки.

Git установлен и по минимуму настроен, можно приступать к его изучению.

Вопросы для самопроверки:

1. Что делает команда `git config --list --show-origin`?

   Варианты ответов:

   1.1. Выводит список текстовых редакторов, доступных для интеграции с Git, и их расположение на машине.

   1.2. Выводит список параметров Git, которые можно настроить.

   1.3. Выводит список настроенных параметров Git с их значениями и сведениями о том, где они хранятся.

   1.4. Выводит список репозиториев, находящихся под контролем Git на данной машине, с указанием на авторов.

2. Для чего при настройке имени пользователя указывается флаг `--global`?

   Варианты ответов:

   2.1. Для указания использовать данное имя пользователя для некоторых репозиториев под управлением Git на данной машине.

2.2. Для указания использовать данное имя пользователя для всех репозиториев под управлением Git на данной машине.

2.3. Для указания использовать данное имя пользователя для работы с Git на данной машине под данной учётной записью.

2.4. Для указания данного имени пользователя при обращении в техподдержку Git.

3. Что из перечисленного НЕЛЬЗЯ использовать как редактор для интеграции с Git?

   Варианты ответов:

   3.1. Vim

   3.2. Emacs

   3.3. Nano

   3.4. ESLint

§ 1.5. Что такое репозиторий и как его создать

Центральная тема, вокруг которой крутится любая система контроля версий – это репозиторий. Что же это такое?

Простыми словами, репозиторий проекта – это папка, в которой хранится ваш проект. Но не просто папка, а такая папка, в которой, помимо файлов и других папок, хранится также история их изменений. Мы можем для каждого файла, лежащего в репозитории, посмотреть, когда и кем он был создан, какие изменения в нём происходили и в какой последовательности, получить более раннюю версию файла и так далее.

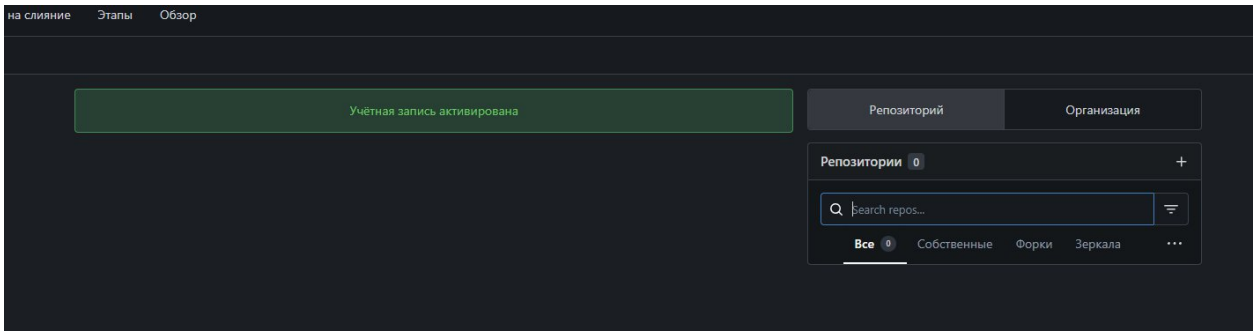
Репозитории могут храниться локально (на вашей машине) и удалённо (на облачном сервисе, на котором работает соответствующее программное обеспечение – как GitHub или GitLab). Во многих случаях (практически во всех случаях командной работы и в большинстве случаев индивидуальной) локальные репозитории разработчиков связаны с удалённым репозиторием (а иногда и не с одним, чтобы обеспечить бесперебойную работу команды в случае отказа облачного сервиса). Соответственно, создать репозиторий можно как локально, так и удалённо, в зависимости от того, что будет удобнее в данном конкретном случае. Можно создать локальный репозиторий (для этого служит команда `git init`), поработать с ним, затем создать пустой удалённый репозиторий и загрузить в него текущее состояние локального репозитория. Однако, гораздо чаще используется другой сценарий – когда репозиторий создаётся удалённо, а затем участники команды клонируют его, то есть, создают локальные репозитории, загружают в них текущее состояние удалённого репозитория и устанавливают связь между удалённым и локальным репозиториями (для этого служит команда `git clone`). Мы будем рассматривать оба сценария, но начнём с более распространённого. В этом случае технический лидер проекта создаёт удалённый репозиторий на сайте, а участники команды (включая и самого лидера) клонируют его локально.

Для скриншотов мы будем в качестве облачного решения рассматривать сервис на базе программного пакета Gitea. Все рассматриваемые действия (клонирование, создание веток, запросы на слияние и т.д.) в GitHub, GitLab и других платформах могут выглядеть по-другому в части расположения и названий кнопок или второстепенной терминологии (так, группы проектов в GitLab будут в Gitea называться организациями), но логика работы останется той же. В качестве упражнения вы можете выполнить одни и те же предлагаемые в учебнике действия на разных открытых платформах (GitHub, GitLab, МосХаб и других) и сравнить интерфейсы.

Откройте в браузере сайт какого-нибудь публичного облачного репозитория. Вы увидите страницу авторизации. Зарегистрируйтесь (или, если вы уже зарегистрированы на этом сервисе, то войдите) с её

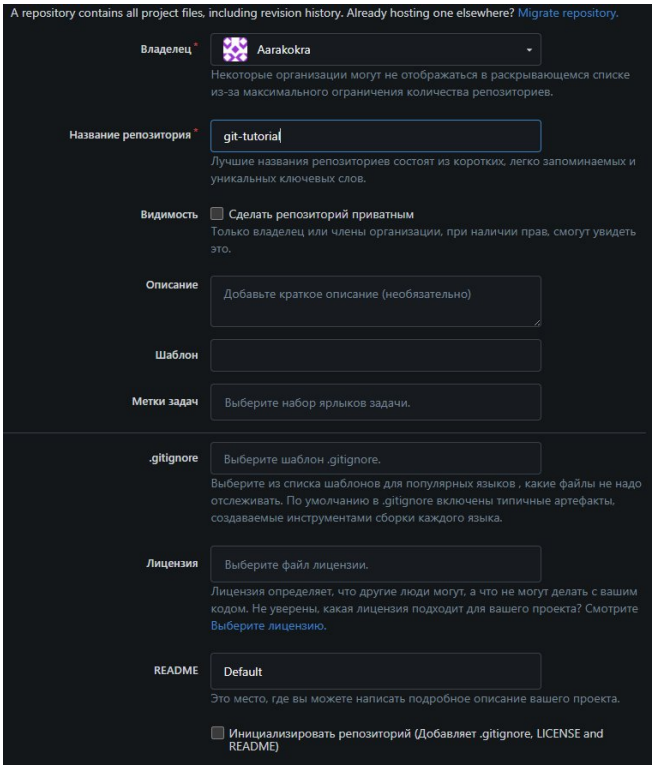
помощью. Если вы этого ещё не сделали, заполните личную информацию – ФИО, e-mail, телефон, пароль и т.д. Возможно, сервис затребует подтверждение e-mail, прежде чем разрешит доступ нового пользователя, или потребует создать второй фактор аутентификации.

В конце концов, так или иначе, вы попадёте на страницу своих репозиториев:



Страница репозиториев нового пользователя в Gitea.

Справа от слова «Репозитории» (в облачных сервисах на других платформах в другом месте) вы увидите кнопку «+» (в других решениях она может называться «Новый репозиторий», «Новый проект» или как-нибудь ещё). После нажатия на эту кнопку сервис может предложить выбор из нескольких возможностей (создание проекта или репозитория с чистого листа, создание из шаблона, импорт из другого облачного решения). В Gitea сразу выходит окно создания нового репозитория, с дополнительной ссылкой Migrate repository для случая, если мы хотим мигрировать репозиторий с другого сервиса:



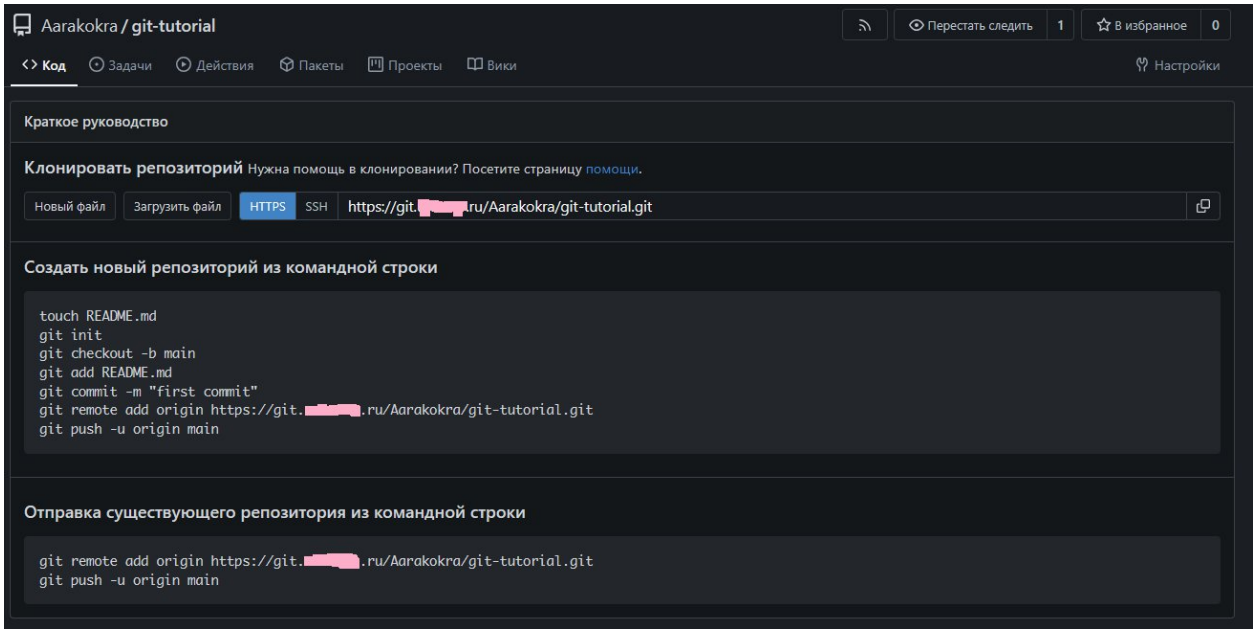
Какая бы система ни использовалась, при создании пустого репозитория система прежде всего предложит нам его назвать. Пожалуйста, избегайте в

названии пробелов и букв национальных алфавитов – на некоторых системах это может привести к нежелательным сложностям. Вместо русских (или других национальных) букв используйте более или менее общепринятую транслитерацию, а вместо пробелов – знаки минуса или подчёркивания (для репозиториев чаще используются знаки именно минуса, т.н. kebab case).

На этой же странице вы можете выбрать уровень видимости репозитория (публичный или приватный; некоторые платформы добавляют и иные уровни видимости).

Также во многих платформах предлагается выбрать файл .gitignore (список игнорирования – мы позже будем изучать работу с ним подробнее), специфичный для разных сред разработки, файл лицензии, под которой будет выпускаться проект, и файл описания проекта README.md (расширение md означает Markdown – текстовый формат, допускающий простые метки форматирования). Но эти настройки в Gitea применятся лишь в том случае, если мы отметим флажок «Инициализировать репозиторий» – в некоторых случаях, как мы увидим позже, важно, чтобы репозиторий был пустым.

После того, как мы указали всё, что нужно, можно нажимать кнопку «Создать репозиторий» или «Создать проект», смотря как она называтеся в выбранной вами платформе. Сервис создаст новый репозиторий, и веб-интерфейс перейдёт на отображение основной страницы репозитория.



Теперь мы можем склонировать репозиторий на свою машину. В некоторых сервисах для этого служит кнопка «Клонировать», при нажатии на которую появляется меню выбора протокола. В сервисах на базе Gitea кнопки выбора протокола (HTTPS или SSH) доступны сразу. Если в выбранном вами сервисе тоже так, попробуйте понажимать на них по очереди, и вы увидите, что URL рядом с ними изменяется. Это не случайно, поскольку именно в URL содержится указание вашему локальному Git на протокол, который ему следует использовать.

Как мы видим, нам, в общем и целом, предлагается два протокола, при помощи которых локальный Git может связаться с удалённым репозиторием: это SSH и HTTPS. В чём разница? Для пользователя разница состоит в основном в том, как сервис будет убеждаться, что мы – это действительно мы. Или, говоря другими словами – в том, как будет проходить аутентификация пользователя.

Для того, чтобы склонировать себе публичный репозиторий, в принципе, аутентификация не нужна. А вот если мы что-то отредактировали и хотим наши изменения предложить владельцу репозитория – тут уже сервис должен знать, кто автор предлагаемых изменений. Что же касается приватного репозитория, тут сервис должен нас аутентифицировать ещё на этапе клонирования, иначе он не сможет дать нам доступ к репозиторию.

Самый простой способ убедиться, что пользователь действительно тот, за кого себя выдаёт – это запросить имя пользователя и пароль. Именно так в простейшем случае на некоторых платформах и производится аутентификация по HTTPS: наш локальный Git стучится на удалённый сервер, тот запрашивает логин и пароль, наш Git их запрашивает у нас и передаёт серверу, после чего сервер, убедившись, что мы – это мы, даёт нашему Git доступ к удалённому репозиторию. Однако, в чистом виде такой способ аутентификации небезопасен – за время, которое проходит между сменами пароля, его можно подобрать перебором, его можно подглядеть при вводе с клавиатуры, его можно перехватить кейлоггером (программой, отслеживающей нажатия на клавиатуру) или сниффером (программой, отслеживающей передачу данных) и т.д. Поэтому многие сервисы облачных репозиториев даже запрещают аутентификацию по логину и паролю, так что клонирование с использованием логина и пароля через HTTPS мы рассматривать не будем.

Несколько улучшает ситуацию с безопасностью доступа по HTTPS использование не пароля, а токена, или PAT (personal access token), который пользователь может создать в настройках своей учётной записи. Токен – это тоже как бы пароль, но:

- у него ограниченный срок действия;
- можно дать предъявителю токена права не на любые действия с любым репозиторием, а на строго ограниченный набор действий со строго ограниченным набором репозиториев;
- поскольку токен обычно не вводится каждый раз руками, а записывается в настройки локального Git, его можно сделать значительно длиннее и «случайнее», чем обычный пароль пользователя.

Вместе с тем, поскольку при каждом сеансе связи локального Git с удалённым сервисом на протяжении срока действия токена этот токен не меняется, проблема подбора перебором или перехвата сниффером остаётся. Поэтому с течением времени появился третий способ аутентификации через HTTPS, известный как

OAuth2. При его использовании Git, видя по предлагаемой URL, что ему нужна авторизация по HTTPS, проверяет, есть ли у него данные для аутентификации. При первом входе их заведомо нет, и тогда Git обращается к специальной вспомогательной программе – под Windows это обычно входящий в комплект инструмент Git Credential Manager (и далее в тексте мы будем ссылаться на него), под Linux обычно используется пакет libsecret – чтобы эта программа выдала ему аутентификационные данные. Вспомогательная программа, в свою очередь, генерирует уникальный случайный код и специальную ссылку, которую открывает в вашем браузере. Вы увидите внезапно открывшийся браузер со страницей выбранного вами сервиса и надписью наподобие «Приложение Git Credential Manager запрашивает доступ к вашим репозиториям». После этого вы проводите аутентификацию именно от своего лица и именно через браузер, как вы бы это делали при обращении к сайту своего репозитория (а если вы уже залогинены, то этого может и не потребоваться, нужно будет просто подтвердить, что действие выполнено вами). Облачный сервис, таким образом, убедится, что это действительно вы. После этого он перенаправит браузер на особую одноразовую страничку, которая невидимо для вас передаст во вспомогательную программу одноразовый код авторизации. Эта вспомогательная программа передаст в облачный сервис уже одноразовый код авторизации (что позволит ей не запрашивать у вас ваши логин и пароль) и получит от него в обмен так называемый токен доступа. Обычно он имеет формат, называемый JWT (JSON Web Token), и выглядит, как некий шифрованный текст. Так оно и есть. В этом токене зашифрованы данные, позволяющие Git взаимодействовать с облачным сервисом от вашего имени, а также уровень доступа и время, в течение которого токен действителен. В дальнейшем Git, видя URL, ведущие на тот же облачный сервис, будет использовать этот токен, пока не истечёт срок его действия, а затем запросит новый.


под капотом

Иногда при этом пользователь, вызвавший какую-либо из команд связи Git с облачным репозиторием, видит ошибку аутентификации, которая при повторе команды пропадает. Это связано с механизмом запроса нового токена. На самом деле, Git Credential Manager получает не один, а два токена – токен доступа (Auth Token) и токен обновления (Refresh Token). Токен доступа используется при каждом обмене между локальным и облачным репозиториями, поэтому время его жизни должно быть ограничено, чтобы уменьшить вероятность подбора. Токен обновления используется только для обновления и сразу заменяется, поэтому время его жизни может быть, в принципе, сколь угодно большим. Когда локальный Git пытается соединиться с облачным репозиторием, используя устаревший токен доступа, он, что закономерно, получает ошибку аутентификации, и пользователь её и увидит. Но Git Credential Manager при этом получает сигнал, что токен устарел и нуждается в обновлении – он, уже сам, соединяется с облачным сервисом, который ведает выдачей токенов, предъявляет токен обмена и в ответ получает новый токен доступа и новый токен обмена, поэтому при следующем выполнении той же команды Git приходит к успеху.

Таким образом, это достаточно безопасный протокол обмена.

Но не все платформы умеют работать по этому способу аутентификации (скажем, Gitea «из коробки» умеет, а МосХаб на момент написания учебника не умеет), и на таких платформах аутентификация по HTTPS обычно отключается администраторами – выбрать её можно, а воспользоваться нельзя. Тогда остаётся аутентификация по SSH, и в дальнейшем мы будем ориентироваться на неё.

При аутентификации по SSH для подтверждения аутентичности пользователя используется асимметричное шифрование. В чём состоит его идея?

В примитивных видах шифрования, которые вы, возможно, встречали в литературе или в школьных задачах по информатике – таких, как шифр Цезаря (буквы сдвигаются по алфавиту на определённое количество символов) или атбаш, известный также как простая литорея или тарабарская азбука (согласные буквы пишутся в ряд слева направо от «Б» до «Н», затем во втором ряду пишутся справа налево от «П» до «Щ», и при шифровании/дешифровании каждая согласная из одного ряда заменяется на согласную из того же столбца другого ряда) – и для шифрования, и для дешифрования используется один и тот же ключ (скажем, для шифра Цезаря – число, указывающее, на сколько позиций нужно сместить каждую букву). Это так называемое симметричное шифрование. При использовании симметричного шифрования у отправителя и у получателя сообщения должен быть в наличии общий ключ (хотя способы его применения для шифрования и для дешифрования могут быть разными).

Как можно было бы решить задачу аутентификации пользователя при помощи симметричного шифрования? Например, так. Пусть в профиле пользователя хранится ключ шифрования. Тогда можно было бы поступать следующим образом: когда наш локальный Git пытается аутентифицироваться на сервисе удалённых репозиториев, сервис мог бы отправлять ему какое-нибудь сообщение, локальный Git бы его шифровал и отправлял обратно. Сервис расшифровывал бы это сообщение при помощи ключа шифрования, который хранится в нашем профиле, и проверял, совпал ли результат с отправленным. Если совпал, значит, сообщение было зашифровано именно тем ключом, который хранится в нашем профиле, а значит, локальный Git имеет доступ к этому ключу и, следовательно, принадлежит нам.

Это, конечно, хорошо, но есть у такой схемы недостаток. Получается, мы должны передать по сети некую информацию, которая в дальнейшем будет служить доказательством того, что мы – это мы. Более того, эта информация потом будет храниться где-то на диске сервера, на котором работает сервис. Что будет, если кто-то получит доступ к этому диску?

Таким образом, аутентификация при помощи симметричного шифрования уязвима минимум дважды – при передаче ключа и при его хранении.

И вот тут на сцену выходит асимметричное шифрование. При асимметричном шифровании используется такой алгоритм, при котором у каждого участника имеется два ключа. Если сообщение зашифровано одним ключом, расшифровать его можно только вторым ключом, первый не подойдёт. И наоборот, если сообщение зашифровано вторым ключом, расшифровать его можно только первым ключом. При этом, зная один ключ, вычислить по нему другой нельзя (точнее, можно, конечно, решить эту задачу простым перебором, но длина ключа такова, что время такого перебора заведомо превышает время, на протяжении которого зашифрованная информация сохраняет ценность – а пользователь за это время уже успеет сменить пару ключей).

Ну хорошо, а как асимметричность решает задачу аутентификации?

Пользователь создаёт на своей локальной машине пару ключей. Один из этих двух ключей в дальнейшем он считает закрытым, или приватным, и никуда его не пересылает, никому не даёт. А вот другой ключ он объявляет открытым, или публичным, и спокойно передаёт его по сети на сервис для сохранения в своём профиле.

Когда локальный Git пытается аутентифицироваться от нашего имени на сервисе, сервис, как и было задумано, отправляет ему некоторое случайное сообщение. Наш локальный Git шифрует его нашим закрытым ключом, и отправляет его сервису. Сервис расшифровывает его нашим открытым ключом, который спокойно хранится в профиле совершенно открыто (на то он и открытый ключ), убеждается, что расшифрованное сообщение равно изначальному, а значит, локальный Git имеет доступ к нашему закрытому ключу, а следовательно, принадлежит нам.

В отличие от симметричного шифрования, тут мы не рискуем своей аутентичностью, ни отправляя ключ по сети, ни доверяя его хранение неизвестному компьютеру на неизвестном диске – ведь она устанавливается фактом наличия доступа к закрытому ключу, а он никуда не передаётся и нигде, кроме как у нас, не хранится.

Также это безопаснее, чем аутентификация по токену, и во много раз безопаснее, чем аутентификация по паролю (помимо того, что закрытый ключ не передаётся, так ещё и информация, которой обмениваются локальный Git с сервисом, каждый раз разная, так что даже подслушивание сетевого трафика ничего не даст).

Однако, остаётся ещё одна уязвимость – а что если злоумышленник получит доступ к нашему диску и украдёт файл с нашим закрытым ключом?

Это тоже решаемая задача. Закрытый ключ при создании пары ключей дополнительно шифруется каким-нибудь обычным симметричным алгоритмом, а ключ шифрования мы задаём при генерации ключей в виде т.н. пароля закрытого ключа. Этот пароль вообще нигде не хранится, кроме как в чертогах нашего разума (если только мы не решим его где-нибудь записать) и никуда не передаётся – когда

локальному Git нужен доступ к закрытому ключу, он просто запрашивает у нас пароль к нему. Кстати, тем самым дополнительно удостоверяется, что за компьютером присутствуем именно мы, и команду локальному Git дали именно мы.

Как же так получается, что для шифрования нужен один ключ, а для дешифрования другой? Что это за чудесный алгоритм? Знать это нам не обязательно, так что этот раздел вы можете пропустить, но если вам интересно, обсудим. Сейчас известно несколько таких алгоритмов, среди которых именно для аутентификации по SSH чаще всего используют т.н. шифрование на эллиптических кривых. Но исторически первым был алгоритм RSA (по первым буквам фамилий создателей: Ривест, Шамир, Адлеман). Он дольше работает и требует более длинных ключей, чем шифрование на эллиптических кривых, но вместе с тем его математику проще понять, так что для примера рассмотрим именно его.

Долгое время изучение простых чисел – таких натуральных чисел, которые делятся только на 1 и на себя – было скорее математической забавой, наукой для науки, чем чем-то, имеющим прикладное значение. Много ли вы помните тем из школьного курса знаний – а в большинстве случаев и из институтского – когда бы знание свойств простых чисел имело значение? И вот в 1976 году был придуман алгоритм, перевернувший весь цифровой мир. Благодаря асимметричному шифрованию мы решаем задачу аутентификации не только при работе с Git. Фактически, благодаря ему существуют банковские карты, онлайн-платежи, электронные подписи, электронный документооборот вообще, онлайн-мессенджеры... Перечислять можно долго. Так как же устроен алгоритм RSA?

Пусть у нас имеется некоторое очень большое число N. Функцией Эйлера φ(N) называется количество натуральных чисел от 1 до N - 1, взаимно простых с N (то есть, таких, что, если составить дробь, где числителем будет это число, а знаменателем N, то эта дробь будет несократимой).

Пусть у нас имеется некоторое число e в диапазоне от 1 до φ(N) такое, что оно взаимно простое с φ(N). Зная e и φ(N), мы можем найти такое число d, что остаток от деления d • e на φ(N) будет равен 1 (тогда число d называется обратным к числу e по модулю φ(N), и его можно найти при помощи расширенного алгоритма Евклида – это почти тот же самый алгоритм Евклида, при помощи которого вы в школе находили наибольший общий делитель двух чисел, но с некоторым дополнением).

Так вот. Пусть у нас имеется некоторое сообщение m. Поскольку всё, в том числе текст, в компьютере является числом, то мы можем рассматривать m как число. Пусть m < N (если нет, мы можем разбить сообщение на блоки, каждый из которых меньше N).

Получим некоторое сообщение M таким образом, что возведём m в степень e и возьмём остаток от деления результата на N. То есть,

$M = m^e \bmod N$, где a mod b – это взятие остатка от деления a на b.

Оказывается (это можно достаточно несложно доказать), что если мы повторим эту операцию, но вместо e возьмём d, а вместо m возьмём M, то мы получим обратно исходное m. То есть,

$m = M^d \bmod N$.

Иными словами, если пару e, N мы считаем ключом шифрования для такого алгоритма шифрования, то пара d, N будет ключом дешифрования, и наоборот. То есть, мы действительно получили асимметричное шифрование.

Остаётся вопрос – неужели действительно сложно, зная e и N, получить d? Ведь, если мы знаем φ(N), то мы, как сказано выше, получаем d из e и φ(N) просто по расширенному алгоритму Евклида?

Да, но самое важное тут: «если мы знаем φ(N)». Вся беда в том, что функция Эйлера φ(N) в общем случае вычисляется только перебором чисел от 1 до N - 1. Если число N очень большое, то эта задача за практически допустимое время не решается.

Ну хорошо, а мы-то? Нам-то как получить эту самую φ(N)?

Для этого мы используем не просто любое N. Хитрость в том, как мы выбираем N. Мы берём два очень больших простых числа – назовём их p и q – и вычисляем N как их произведение. В таком случае φ(N) равно просто произведению (p-1)•(q-1) (это тоже достаточно легко доказать). Таким образом, зная простые множители, из которых образовано N, мы практически автоматически получаем φ(N), а значит, легко получаем d из e (ну или наоборот, как вам будет угодно).

Хорошо. А злоумышленник, зная, что N представляет собой произведение двух простых чисел, может же так же легко получить φ(N)?

Так же, да не так же. Для этого ему сначала нужно узнать эти два простых числа. А для этого ему нужно перебрать простые числа от 2 до квадратного корня из N. Что, при условии, что N действительно очень большое, опять же, задача, решаемая за практически нецелесообразное время.

Таким образом, данный алгоритм действительно решает все поставленные задачи: мы легко можем получить пару ключей, но по одному ключу второй ключ восстановить нельзя иначе как перебором.

Для того, чтобы наш локальный Git смог аутентифицироваться по SSH, нам нужно выполнить три шага:

1. Нужно создать пару ключей SSH.
2. Нужно передать в облачный сервис наш публичный ключ.
3. Нужно указать Git, какой из закрытых ключей использовать с облачным сервисом (да-да, даже если у нас только одна пара ключей, без этого

зачастую – особенно в Windows – аутентификация по SSH может не работать).

Итак, начнём с генерации пары ключей. Для этого нам понадобится утилита командной строки ssh-keygen. Под Linux она уже предустановлена в подавляющем большинстве дистрибутивов. Если вдруг у вас не так, установите её при помощи своего менеджера пакетов: в yum будет пакет openssh, в apt будет пакет openssh-client. Под Windows 10 утилита вошла в предустановленный набор в релизе 1809 (осенний релиз 2018 года), под Windows 11 входит изначально. Если вдруг ваша система древнее (что не рекомендуется, но мало ли, какие у вас обстоятельства), установите дополнительный компонент OpenSSH Client.

Команду ssh-keygen, в принципе, можно выполнить и без параметров, и результат, в принципе, устроит. Но чтобы сделать сразу хорошо, мы задействуем ключи -t для задания алгоритма шифрования и -f для указания имени файла закрытого ключа.

Как уже упоминалось, на настоящий момент есть несколько алгоритмов асимметричного шифрования, и в первую очередь это рассмотренный нами как пример RSA и несколько алгоритмов, использующих т.н. эллиптические кривые. На настоящий момент к использованию рекомендуется алгоритм на эллиптических кривых ed25519, который понимают практически все облачные сервисы.

Что касается имени файла закрытого ключа, оно должно быть без расширения, и кроме того, лучше сразу поместить файл в наиболее подходящую для этого папку. В Linux это

```
~/.ssh
```

Что касается Windows, там путь посложнее, но, к счастью, есть сокращение, которое система понимает (строго говоря, это переменная окружения):

```
%USERPROFILE%\.ssh
```

По факту, если все настройки при установке Windows и создании пользователя были взяты по умолчанию, то %USERPROFILE% имеет значение C:\Users\логин, где логин – это логин пользователя (ну или имя учётной записи, как вам будет угодно).

Создадим пару ключей с именем git_tutorial:

```
ssh-keygen -t ed25519 -f ~/.ssh/git_tutorial
```

Здесь взята нотация Linux. Если вы выполняете эту команду в Windows, но из Git Bash, тут ничего менять не надо, а если из обычной командной строки Windows, она будет выглядеть

```
ssh-keygen -t ed25519 -f %USERPROFILE%\.ssh\git_tutorial
```

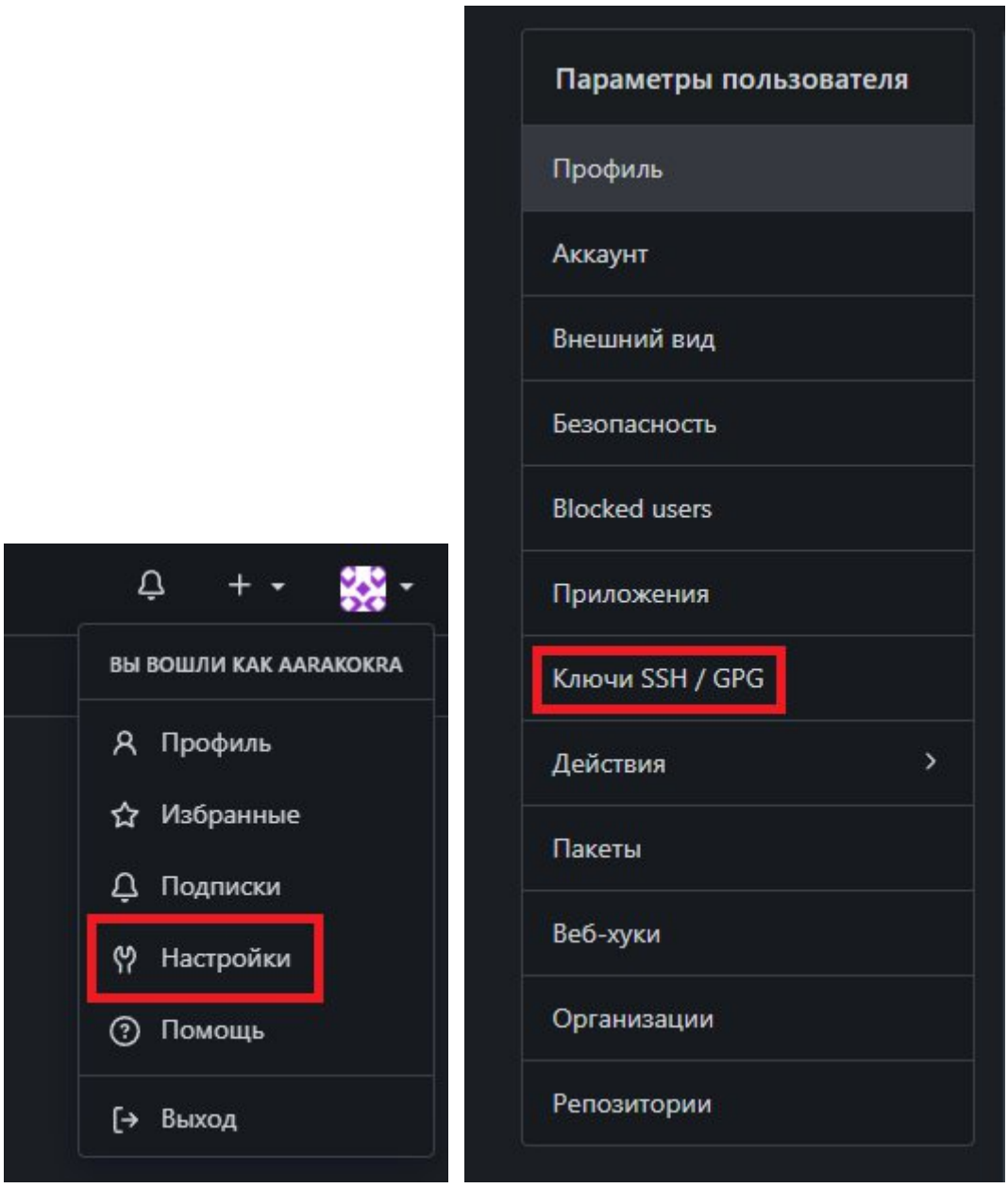
В процессе генерации пары ключей ssh-keygen предложит задать пароль. Как вы наверняка помните, мы уже говорили, что у нас есть возможность защитить

паролем файл с закрытым ключом, чтобы его кража не помогла злоумышленнику. Обратите внимание, что процесс ввода пароля никак не отображается на экране! Это делается для того, чтобы, подглядывая на экран (очно или через программу перехвата изображения), нельзя было определить количество символов пароля. После того, как пароль введён, его нужно будет повторить – это стандартная страховка от ошибки ввода. Если вы не хотите защищать закрытый ключ паролем, можете при запросе пароля просто сразу нажать <ENTER>, этот шаг будет пропущен. Тогда при использовании закрытого ключа Git не будет запрашивать пароль, но сам ключ при этом не будет защищён от кражи.

После того, как пароль введён и продублирован (либо после пропуска этого шага), ssh-keygen создаёт файлы закрытого ключа (с указанным в ключе -f именем) и открытого ключа (к указанному имени добавляется расширение .pub, от английского «public»). На экран утилита выводит «отпечаток ключа» (по-английски «отпечаток пальца» - «fingerprint») – некоторым образом вычисленная последовательность байт (хэш-код), которую проверить на совпадение проще, чем ключ целиком. Кроме того, утилита строит странную псевдографическую картинку – randomart. И отпечаток ключа, и рандомарт служат для того, чтобы защитить открытый ключ от подмены – всегда можно скачать обратно открытый ключ, посчитать по нему отпечаток (для автоматического сличения) или рандомарт (для сличения глазами – соответствие картинок зрением устанавливается быстрее, чем соответствие длинной строки символов) и убедиться, что злоумышленник не подменил открытый ключ. Кроме того, можно опубликовать их отдельно и оставить в своём профиле ссылку на них – с той же целью.

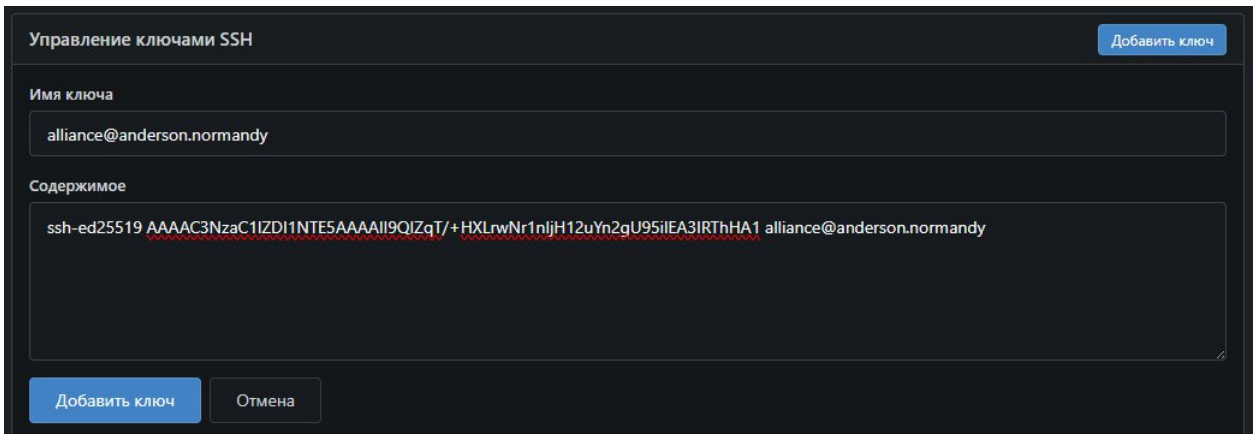
Первый шаг выполнен – пара ключей создана. Теперь нужно выполнить второй шаг – передать открытый ключ в облачный сервис. Для примера опять же будем использовать сервис на базе Gitea – на других сервисах всё делается аналогично (хотя пункты меню могут немного отличаться по названию и расположению).

Заходим на сервис, после чего переходим в свой профиль (на платформе Gitea – в настройки профиля) и выбираем группу «Ключи SSH»:



Возможно, для появления поля, куда можно ввести ключ, потребуется нажать кнопку с названием наподобие «+», «Добавить ключ» и т.д. После этого должны появиться поля для самого ключа, для его названия и, на некоторых платформах, для срока действия.

В любом редакторе (например, в блокноте) открываем созданный утилитой **открытый** ключ (файл с расширением pub), копируем его содержимое и вставляем его в поле «Ключ», «Содержимое» или как-то ещё, в зависимости от платформы:



Обратите внимание на устройство содержимого файла открытого ключа. Оно состоит из трёх частей. Сначала идёт указание на используемый алгоритм шифрования, чтобы тот, кто использует открытый ключ (в данном случае облачный сервис) знал, каким алгоритмом следует расшифровывать пришедшее от локального Git сообщение для проверки его совпадения с отправленным. Далее идёт собственно ключ. Наконец, в конце идёт комментарий, который добавляет утилита генерации – предлагаемое по умолчанию имя ключа. Утилита его формирует из учётной записи пользователя, сгенерировавшего ключ (если используется доменная система, то с доменом вместе) и имени компьютера, на котором был сгенерирован ключ, в его локальной сети.

Как только вы вставили открытый ключ в нужное поле, комментарий этого ключа копируется в поле «Название» (или «Имя», или аналогично). Вы можете задать ключу другое название – это не влияет ни на что, кроме того, как ключ будет отображаться на странице в списке зарегистрированных на сервисе открытых ключей пользователя (каждый пользователь может зарегистрировать больше одного открытого ключа – например, чтобы не таскать закрытый ключ с одной машины на другую, пользователь может на каждой создать свою пару ключей).

Для повышения безопасности на некоторых платформах можно ограничить срок действия ключа, введя дату окончания его действия в поле «Срок действия».

После того, как все настройки выполнены, можно зарегистрировать вставленный открытый ключ в своём профиле, нажав кнопку «Добавить ключ».

Все действующие ключи отображаются на этой странице в разделе «Ваши ключи SSH», расположенном ниже кнопки «Добавить ключ». Любой из этих ключей можно удалить досрочно при помощи соответствующей кнопки.

Осталось настроить наш локальный Git так, чтобы он гарантированно подхватывал наш закрытый ключ при аутентификации на облачном сервисе. Идём в папку ~/.ssh (если вы на Linux) или %USERPROFILE%\.ssh (если вы на Windows) и открываем (если его нет, то создаём) файл config (именно так, без расширения).

Файл ~/.ssh/config содержит настройки клиента SSH вообще (он используется не только для Git). Все его параметры мы разбирать не будем, ограничимся только необходимыми.

Находим в файле строчку

```
Host сайт хоста
```

Сайт хоста здесь будет hub.mos.ru для МосХаба, может быть как-то наподобие git.corp.local для локального корпоративного сервиса, и так далее.

Если такой строчки нет, создаём её (обратите внимание, что перед словом «Host» не должно быть пробелов).

В одной из последующих строк (но до следующей строки Host, если таковая есть) ищем строку, которая начинается с одного или нескольких пробелов и строки «IdentityFile». Если такой строки нет, создаём её:

```
  IdentityFile ~/.ssh/git_tutorial
```

Сначала идёт несколько пробелов (обычно используют два), затем ключевое слово IdentityFile, опять пробел, а затем имя файла **закрытого** ключа вместе с путём к нему. Обратите внимание, что путь вводится в нотации Linux, вне зависимости от того, в какой системе вы работаете.

Если такая строка (IdentityFile под нужным Host) существовала, убедитесь, что в ней проставлен именно тот файл закрытого ключа, который соответствует отправленному в сервис открытому ключу, иначе аутентификация не получится. Если имя ключа оставлено стандартным (id_rsa или id_ed25519, в зависимости от алгоритма шифрования), скорее всего, параметр IdentityFile и вовсе не понадобится – но следует это проверить для вашего конкретного случая.

Итого, у вас должно получиться что-то такое:

```
Host git.mysuperpuperhost.ru
  IdentityFile ~/.ssh/git_tutorial
```

Если вы работаете с несколькими сервисами, вы можете, таким образом, для каждого из них указать закрытый ключ. Это может быть как один ключ на несколько сервисов, так и по своему ключу для каждого из них.

Мы настроили аутентификацию нашего локального Git на выбранном облачном сервисе. Теперь мы можем склонировать наш репозиторий, используя аутентификацию через SSH.

Зайдём на страницу нашего проекта и нажмём на кнопку «Клонировать», если она есть, или нажмём на кнопку SSH, если кнопки HTTPS и SSH доступны сразу.

Нам нужен URL, который нам покажет облачный сервис после выбора протокола SSH. Обычно этот URL для SSH выглядит как

```
  git@сайт:пользователь/репозиторий.git
```

Для HTTPS, кстати, он будет выглядеть, скорее всего, как

```
  https://сайт/пользователь/репозиторий.git
```

В командной строке (лучше, если это будет командная строка GitBash) зайдём в папку, куда мы хотим склонировать репозиторий, и выполним команду git clone, которой передадим в качестве параметра строку из поля рядом с кнопкой SSH (или имеющейся в меню, если таковое есть, по пункту «Клонировать с помощью SSH»):

```
git clone git@git.somehost.ru:итакдалее
```

Локальный гит запросит пароль к закрытому ключу, если мы защитили его паролем. Если нет, то сразу склонирует репозиторий – в папке, в которой мы находимся, создаст подпапку с именем проекта (которая отныне будет папкой проекта и его локальным репозиторием), в ней создаст служебную скрытую папку .git (её содержимое нас на данном этапе не касается), скачает последние версии всех находящихся в репозитории файлов (если таковые есть) со всей структурой подпапок и запишет всё это в локальный репозиторий (на самом деле, под капотом прячется больше работы – история изменений файлов тоже сдублируется в локальный репозиторий).

Следует учесть, что порт 22, который служит для взаимодействия по протоколу SSH по умолчанию, на частных или внутренних корпоративных серверах Git может использоваться для других целей (например, для удалённого администрирования сервера, также по протоколу SSH), и работа Git через SSH может осуществляться по другому порту (например, 2222). Если просто добавить номер порта к имени хоста, как это обычно делается в URL для браузера (`git clone git@gitlab.corp.local:2222/group/project.git`), Git может воспринять (и скорее всего воспримет) этот номер как часть пути. Чтобы этого не произошло, требуется в URL для Git явным образом указать протокол:

```
git clone ssh://git@gitlab.corp.local:2222/group/project.git
```

Однако есть способ проще. Для этого в настройках в ~/.ssh/config (при работе в Windows через обычную командную строку или PowerShell этот путь следует указывать как %USERPROFILE%\.ssh\config) для такого хоста нужно явным образом указать порт:

```
Host gitlab.corp.local
  IdentityFile ~/.ssh/git_tutorial
  Port 2222
```

После этого в URL для команды клонирования порт можно не писать.

Кстати, обратите внимание, что этот файл config находится не в какой-то из папок, специфичных для Git. Папка .ssh специфична для настроек протокола SSH вообще в целом. Поэтому содержащиеся там настройки будут использоваться не только Git, а любой программой, которая использует протокол SSH: VS Code, SourceTree, различными клиентами SSH и агентами непрерывной интеграции кода. Таким образом, настройка, сделанная один раз, будет работать везде.

Служебная скрытая папка .git содержит «внутреннюю кухню» Git. Начинающий пользователь (да даже и опытный в подавляющем большинстве случаев) не

получит никакой пользы от чтения её содержимого, а при попытке его модификации может необратимо повредить репозиторий. Все изменения в содержимое данной папки должны вноситься только через команды Git.

Получившийся локальный репозиторий содержит, помимо прочего, ссылку на облачный репозиторий, с которого он склонирован. Чтобы убедиться в этом, используем команду

```
git remote –v
```

«Remote» по-английски и означает «удалённый» (не в том смысле, что кто-то его удалил, а в том смысле, что он может быть в удалении от нас). Ключ –v в данной команде означает «verbose», т.е., «подробный». Если мы не будем использовать этот ключ, а просто введём команду git remote, локальный Git выдаст нам только имена удалённых репозиториев, с которыми связан наш локальный репозиторий (в данном случае, скорее всего, вы увидите только слово origin). С ключом –v вы, скорее всего, увидите две строчки, начинающиеся со слова origin, за которым следует та строчка, которую вы копировали с сайта облачного сервиса для вставки в команду git clone, а под конец «(fetch)» в одной строке и «(push)» в другой.

Слово origin – это имя, под которым локальный Git запомнил ссылку на удалённый репозиторий. Чаще всего локальный репозиторий связывают только с одним удалённым и дают этой связи имя origin. Однако при желании можно связать один локальный репозиторий с несколькими удалёнными (что застрахует нас от внезапной потери удалённого репозитория, но придаст немного сложности синхронизации данных) или переименовать связь.

Строчка с сайта после имени связи – это ссылка особого вида, по которой локальный Git понимает, куда идти для скачивания свежих изменений репозитория или для их закачивания, как авторизовываться и как найти нужный проект в удалённом сервисе.

Наконец, слова «fetch» («извлечь») и «push» («затолкать») указывают, для каких действий актуальна ссылка. Таким образом, мы видим, что локальный Git зарегистрировал связь созданного локального репозитория с облачным репозиторием, ссылку на который мы скопировали с сайта, под именем origin, как для извлечения с облака свежих изменений кода, так и для отправки на облако своих правок.

Чаще всего именно так репозитории и создаются – сначала создаётся облачный репозиторий на выбранном сервисе, затем все участники команды его клонируют себе. Однако, позже мы будем рассматривать и второй способ – когда сначала руководитель проекта создаёт репозиторий локально, выполняет все нужные первоначальные действия (например, настройки, генерацию начальной версии кода и т.д.), и только затем создаёт облачный репозиторий и отправляет в него созданный локальный репозиторий. И конечно же, мы рассмотрим, как отправить в облако сделанные изменения, и как их оттуда получить.

НА ЗАМЕТКУ:

1. SSH – сокращение от Secure SHell («безопасная оболочка»). Изначально протокол создан для удалённого доступа к машинам (обычно серверам).
2. Ключ SSH можно также использовать для электронной подписи коммита (хотя чаще для этого используется другой вариант – ключ GPG). На настоящий момент это поддерживают лишь очень немногие облачные сервисы, поэтому пока мы эту возможность изучать не будем.

Вопросы для самопроверки:

1. Что такое репозиторий?

   Варианты ответов:

   1.1. Инструмент автоматического форматирования кода.

   1.2. Папка, в которой хранятся файлы проекта и история их изменений.

   1.3. Система резервного копирования.

   1.4. Платформа для совместной разработки.

2. Какой файл GitHub **не** предлагает создать при создании репозитория?

   Варианты ответов:

   2.1. README

   2.2. .gitignore

   2.3. Файл лицензии

   2.4. package.json

3. При помощи какой команды можно склонировать облачный репозиторий?

   Варианты ответов:

   3.1. git clones

   3.2. git cloned

   3.3. git clone

§ 1.6. Базовый цикл работы с Git

Итак, репозиторий мы создавать научились. Мы уже знаем, что он отличается от обычной папки тем, что позволяет хранить историю изменения содержимого. Из прочитанного вы уже, конечно, поняли, что эта история хранится не в виде непрерывного «кино», а в виде серии как бы моментальных снимков содержимого в определённые моменты – как если бы мы время от времени создавали копии папки проекта. Такие «моментальные снимки» называются коммитами. В этом параграфе мы, собственно говоря, и научимся создавать коммиты, просматривать их историю и сравнивать коммиты между собой.

Мы уже умеем клонировать существующий репозиторий из облачного сервиса, но сейчас рассмотрим второй путь создания репозитория – когда он создаётся на локальной рабочей машине.

Для начала создадим локальный проект, для которого мы в дальнейшем будем создавать репозиторий. Для примера мы создадим папку с названием, например, MyAwesomeCoolSite (или любым другим) и в ней несколько файлов: допустим, README.md для описания проекта, index.html как главная страница сайта, styles.css как таблица стилей сайта и script.js как скрипты, оживляющие фронтенд.

Если вы работаете под управлением Windows, запустите Git Bash. Возможно, после установки Git у вас даже появился пункт в контекстном меню Проводника: «Открыть Git Bash в этой папке», «Open Git Bash here» или как-то ещё. Если его нет, вы можете запустить Git Bash и перейти в нужную папку при помощи команды cd (от английского change directory). Имейте в виду, что Git Bash использует для путей в файловой системе нотацию Linux. Так, для перехода в подпапку Sources\Web домашней папки (в которой расположены также папки «Мои документы», «Моя музыка» и другие), вам понадобится развернуть слэши в другую сторону и указать явно, что путь начинается от домашней папки:

```
cd ~/Sources/Web
```

а если папка Sources при этом находится не в домашней папке, а, скажем, в корне диска D:\, то вместо указания диска в привычном для Windows стиле с двоеточием вы должны указать его как папку, находящуюся в корне файловой системы:

```
cd /d/Sources/Web
```

Если вы работаете под управлением Linux, откройте терминал и перейдите в нужную папку.

Для создания новой папки в терминале Linux (и в Git Bash) используется команда mkdir:

```
mkdir имя_папки
```

Создайте папку MyAwesomeCoolSite (или другое название). Пока это просто папка. Если мы меняем содержимое какого-то файла, мы уже не можем получить информацию о том, что именно было изменено – ведь контроля версий нет. Чтобы включить контроль версий, нужно превратить эту папку в репозиторий. Но сначала нужно перейти в неё (иначе репозиторием станет не сама папка, а та папка, которая её содержит). Перейдите в папку MyAwesomeCoolSite (или как ещё её назвали) командой

```
cd имя_папки
```

Чтобы превратить папку в репозиторий, служит команда:

```
git init
```

Выполните эту команду. Если всё было сделано правильно, появится примерно такая надпись:

```
Initialized empty Git repository in ваша папка/.git
```

При создании репозитория Git создаёт в папке проекта скрытую подпапку с именем .git (обратите внимание на точку перед именем папки – в Linux имена, начинающиеся с точки, делают папку скрытой автоматически, в Windows для этого устанавливается специальный атрибут папки в файловой системе, Git его проставляет самостоятельно).

В Linux для просмотра содержимого папки служит команда ls (от английского «list» – «список»), в Git Bash она тоже действует, но действует и её аналог для Windows – команда dir. Обе эти команды по умолчанию не показывают скрытые папки:

```
$ ls
README.md  index.html  script.js  styles.css
```

Чтобы включить их отображение, используется ключ -a (от английского «all», т.е. «все»):

```
$ ls –a
.  ..  .git  README.md  index.html  script.js  styles.css
```

Мы видим, что, действительно, папка .git была создана. Как и в прошлом параграфе, мы не должны вмешиваться в её содержимое, и тем более не должны удалять эту папку. В противном случае папка перестанет быть репозиторием, а вся история изменений будет потеряна (если только мы не успели всю её сохранить на сервис).

Кстати, дополнительно мы видим папки со странными именами «.» и «..». Это стандартные и для Linux, и для Windows обозначения: имя «.» означает просто

информацию о текущей папке, а «..» – информацию о папке, в которой находится текущая.

Как мы уже говорили, репозиторий не записывает каждое изменение прямо в момент его возникновения в хронологическом порядке. Конечно, нет. В репозитории хранится то, как выглядит содержащийся в нём набор файлов и подпапок и содержимое файлов в определённые моменты. Эти моменты мы определяем сами. То есть, мы как бы в определённый момент говорим: «Остановись, мгновенье, ты прекрасно!», и Git делает копию всего содержимого репозитория в особом формате в скрытую папку .git (разумеется, содержимого за исключением самой папки .git). Получается как бы «моментальный снимок» содержимого репозитория (собственно, на самом деле и производится не копия, а снимок – например, неизменённые данные не копируются). Такой снимок и называется коммитом или фиксацией (термин «коммит», транслитерация английского термина «commit», более популярен, но в русскоязычных интерфейсах иногда встречается и термин «фиксация»). Чтобы мы не запутались, Git требует, чтобы мы к каждому коммиту оставляли сообщение – краткую информацию о том, в чём заключаются изменения проекта по сравнению с предыдущим коммитом – а также сохраняет информацию о том, кто выполнил коммит и когда.

В каждый момент мы можем посмотреть состояние репозитория – информацию о том, есть ли в содержимом репозитория изменения по сравнению с прошлым коммитом (новые, изменённые и/или удалённые файлы).

Чтобы посмотреть состояние репозитория, служит команда:

```
git status
```

При выполнении данной команды мы должны находиться в папке репозитория или какой-нибудь из подпапок (разумеется, папка .git и её подпапки не считаются).

Выполните эту команду. Вы увидите что-то в таком духе:

```
$ git status
On branch master


No commits yet


Untracked files:
     перечень файлов
nothing added to commit but untracked files present
```

То есть, Git говорит нам (что бы это ни означало), что мы находимся в ветке «master», коммитов в этом репозитории ещё не было, при этом нет изменений, добавленных в будущий коммит, но имеются неотслеженные файлы.

Что это означает?

Начнём с первой фразы: «On branch master», т.е. «в ветке master». Веткой (branch) в Git называется, как бы это сказать, «рабочее пространство», в котором мы ведём разработку. У каждой ветки могут отличаться набор файлов и папок и их содержимое. Мы можем переключать репозиторий с одной ветки на другую, и содержимое папки проекта соответствующим образом меняется. История изменений тоже может различаться в разных ветках. Ветка с названием master либо main обычно создаётся локальным Git либо облачным сервисом (в зависимости от того, где мы решили создавать проект изначально) по умолчанию, и во многих компаниях или рабочих группах в этой ветке в каждый момент хранится текущая поставляемая пользователю или заказчику версия кода. Для текущей версии в разработке в таком случае создают ветку с названием developer или dev. Таким образом, разработчики могут работать над реализацией новых функций продукта, не боясь фатально сломать поставляемый заказчику код. Иногда, наоборот, разработку ведут в ветке по умолчанию, а для готовых версий продукта, поставляемых в очередном релизе, служит ветка с названием prod, production или trunk. В дальнейших уроках мы будем подробнее (и понятнее, надеюсь) разбирать концепцию веток.

Далее, Git говорит, что коммитов ещё не было. Это действительно так – мы ведь ещё даже не проходили, как их, собственно, делать.

Затем Git перечисляет те файлы, которые уже лежали в нашей папке до того, как она стала репозиторием, и называет их «untracked files» – «неотслеженные файлы». В каком смысле «неотслеженные»?

Файл в репозитории может находиться в одном из следующих состояний:

- Неотслеженный (untracked) – это означает, что этот файл ещё не добавлялся в репозиторий (хотя он и находится в папке репозитория), и Git ещё не производил с ним никаких действий.

- Изменённый (modified) – это означает, что раньше уже был хотя бы один коммит, в котором содержимое этого файла было записано, но сейчас его содержимое отличается от того, которое было на момент последнего коммита (точнее, последнего коммита в ветке, в которой мы сейчас находимся).

- Неизменённый (clean, буквально чистый) – это означает, что файл уже был записан в каком-либо из коммитов (как говорят, **закоммичен**), и его содержимое на текущий момент ровно такое же, какое было на момент последнего коммита.

- Игнорируемый (ignored) – вообще говоря, Git не должен его даже упоминать. Однако, если смотреть состояние репозитория не при помощи команды самого Git (git status), а при помощи какой-нибудь среды разработки или какой-нибудь визуальной программы для работы с Git (например, Tortoise Git), вы, возможно, увидите игнорируемые файлы в каком-нибудь особом виде (в сером шрифте, со специальным значком или ещё как-нибудь). При помощи только командной строки вы такой файл увидите по команде ls, но не увидите по команде git status.

- Удалённый (deleted) – это означает, что на момент последнего коммита он существовал и был записан как часть содержимого репозитория, но с тех пор был удалён.

- Переименованный (renamed) – это означает, что содержимое файла с момента последнего коммита не изменилось (в некоторых случаях Git распознаёт файл как переименованный даже при наличии небольших изменений), но его имя (а иногда и расположение) с тех пор изменилось.

Сделанные в репозитории изменения – наличие неотслеженных файлов, изменения в изменённых файлах и т.д. (фактически, всё, кроме неизменённых и игнорируемых файлов) – не обязательно попадут в очередной коммит. Это потому что изменения также могут быть в одном из нескольких состояний:

- Неотслеженные либо неиндексированные (unstaged) – Git знает об этом изменении, но не будет добавлять его в очередной коммит.

- Отслеженные либо индексированные (staged) – Git добавит это изменение в очередной коммит.

Индексированное состояние, индекс или staging area – это промежуточное состояние между реальным состоянием файлов в папке и состоянием файлов в репозитории на момент будущего коммита. Физически оно реализуется как особый файл в папке .git, в котором хранится информация о том, какие изменения добавлять в будущий коммит. Таким образом, в каждый момент времени есть три состояния файлов в репозитории:

- Файл, каким он был на момент последнего коммита (committed)

- Файл, каким он будет на момент будущего коммита (staged)

- Файл, каким он является по факту в папке на текущий момент (working)

Фактически, все эти три версии могут отличаться друг от друга – если мы, например, закоммитили файл, потом изменили, потом добавили его в индекс, а потом ещё раз изменили. В таком случае в следующий коммит попадёт не текущее содержимое файла, а его содержимое на тот момент, когда файл был добавлен в индекс. Однако, если мы добавили файл в индекс один раз, а потом, не делая коммита, изменили файл и добавили его в индекс ещё раз, то в индексе будет

сохранено только содержимое на момент последнего добавления в индекс, а не обе версии.

Может быть, всё это – ветки, индекс, история коммитов – покажется слишком сложной концепцией, пока мы представляем это в виде текста. Однако, на практике это осваивается в более наглядной форме, поэтому, когда мы приступим к практическим занятиям и лабораторным работам, эти понятия уложатся в голове в стройную картинку... По крайней мере, в идеале должно быть так.

Чтобы добавить какой-либо файл (если он не находится в состоянии «неизменённый» или «игнорируемый») к индексу, используется команда:

```
git add имя файла
```

Мы также можем одним махом добавить в индекс все сделанные на текущий момент изменения в текущей папке и её подпапках, вместо имени файла указав точку:

```
git add .
```

Это потому что и в Windows, и в Linux точка вместо имени файла или папки означает просто текущую папку (а две точки подряд – папку уровнем выше).

Считается, что командой добавления всей текущей папки лучше не злоупотреблять, потому что тогда случайно можно добавить в индекс файл, который добавлять туда не надо (например, мы в порядке эксперимента или для отработки какой-то локальной особенности поменяли настройки проекта, мы хотим их сохранить, но не хотим добавлять в репозиторий, чтобы не сломать настройки у коллег). Но если это всё же случилось, можно удалить из индекса файл, который туда был добавлен (но ещё не закоммичен) командой:

```
git reset имя файла
```

Вообще, хорошая практика, перед тем, как делать коммит, смотреть, какие файлы попали в индекс, а какие нет (это тоже покажет команда git status).

НА ЗАМЕТКУ:

Некоторые файлы вообще никогда не надо добавлять в коммит. Это прежде всего файлы, которые содержат важные для информационной безопасности данные (пароли, закрытые ключи), и файлы, которые перестраиваются автоматически на машине каждого разработчика и могут отличаться от машины к машине (например, среда разработки может хранить там свои настройки, список последних открытых файлов и информацию о том, какие файлы и на каком месте просматриваются в редакторе в данный момент). Также, скажем, при работе с фреймворком Angular автоматически заполняется большим количеством файлов папка node_modules, и хранить её в репозитории нерационально – объём может быть огромным, а заполняется всё равно автоматически. Позже мы подробно рассмотрим, как автоматически исключать такие файлы из рассмотрения Git (для этого служит список игнорирования).

После того, как мы добавили в индекс те (и только те) файлы, изменения в содержимом которых (или изменения в их имени или в самом факте их существования) должны попасть в очередной коммит, мы можем, собственно говоря, выполнить сам коммит при помощи команды:

```
git commit
```

При выполнении этой команды Git откроет тот редактор, который был настроен, чтобы пользователь ввёл сообщение коммита – краткую информацию о том, что было сделано между коммитами. Коммит без сообщения делать нельзя. Если настройка используемого редактора не производилась, Git, скорее всего, откроет Vim. Это весьма удобный редактор в текстовом режиме, но работа с ним довольно сильно отличается от интуитивно привычной, поэтому мы немножко отвлечёмся на базовый минимум.

Файл, открытый в Vim, по умолчанию открывается в так называемом основном режиме. В этом режиме мы не можем вводить текст – клавиши с буквами и знаками препинания трактуются не как текст, который нужно вставить в файл, а как различные команды редактора, такие, как сохранение файла, удаление строки и т.д.

Чтобы начать редактировать файл, нужно перевести Vim в режим вставки текста, нажав в латинской («английской») раскладке клавиатуры клавишу i (от английского «insert» – «вставить»). После этого можно вводить сообщение обычным способом.

Чтобы сохранить сделанные изменения и выйти, нужно, прежде всего, вернуться из режима вставки, нажав <ESC>. После этого нужно перейти в режим ввода спецкоманд, нажав клавишу двоеточия (:). В нижней строке Vim появится символ двоеточия. Затем нужно нажать w (от английского «write» – «записать»), q (от английского «quit» – «уйти») и подтвердить эти команды нажатием <ENTER>.

Или проще говоря:

- Видим открывшийся Vim
- Нажимаем «**i**»
- Вводим сообщение для коммита
- Нажимаем <ESC>
- Нажимаем «**:wq**»
- Нажимаем <ENTER>

Можно ввести сообщение коммита прямо в команде коммита, используя ключ –m (от английского «message» – «сообщение») и кавычки вокруг текста сообщения:

```
git commit –m "текст сообщения"
```

В этом случае Git не будет открывать редактор – ведь сообщение уже есть.

Обычно при работе с Git из командной строки используют –m для коротких сообщений коммита (например, «fix: исправлена надпись на кнопке выхода»), а Vim (или другой заданный в настройках Git редактор) для многострочных сообщений. В последнем случае рекомендуется в первой строке делать резюме изменений, максимально кратко и обще, затем отбить пустую строку, а в последующих строках перечислить изменения более подробно, начиная каждую строку с дефиса и пробела за ним:

```
feature: логин по OpenID


– добавлена авторизация через VK
– добавлена авторизация через ЕСИА
– добавлена авторизация через Сбер ID
```

Можно даже не делать git add, а просто добавить ключ –a (от «add» – «добавить») к сообщению:

```
git commit –a –m "текст сообщения"
```

Однако, неотслеженные файлы при этом добавлены не будут. Кроме того, мы в таком случае теряем ту возможность, которую нам даёт сам факт наличия индекса – возможность посмотреть перед коммитом его будущий состав и, при необходимости, внести коррективы.

Во многих рабочих группах принято начинать сообщение коммита со знака решётки и номера задачи в трекере задач (если трекер задач интегрирован с облачным репозиторием, в задаче при отправке коммита в облако появится комментарий с текстом сообщения и ссылкой на сам коммит, а в сообщении коммита при просмотре его в веб-интерфейсе облачного репозитория – ссылка на задачу в трекере, это очень удобно) и/или с согласованного в рабочей группе кода, максимально кратко описывающего область, в которой произведено изменение: например, «feat(api)» для добавления новой функции в API или «fix(db)» для исправления ошибки в структуре базы данных.

Когда мы, наконец, делаем коммит, Git создаёт новый «моментальный снимок», в котором будут присутствовать те же файлы, которые присутствовали в предыдущем коммите и не были изменены, а для остальных файлов (кроме игнорируемых) в «моментальном снимке» будет присутствовать то состояние, которое записано в индексе. После коммита индекс очищается.

В нашем случае мы превратили папку в репозиторий, когда в ней уже были файлы. Эти файлы у нас ещё не участвовали ни в одном коммите, поэтому они совершенно справедливо помечены Git как неотслеженные.

Добавьте их в индекс и выполните коммит с сообщением, допустим, «Начали работу над проектом». Git отрапортует о проделанной работе:

```
$ git commit
[master (root-commit) 7cf3e89] Начали работу над проектом
 4 files changed, 9 insertions(+)
 create mode 100644 README.md
 create mode 100644 index.html
 create mode 100644 script.js
 create mode 100644 styles.css
```

Мы снова увидим название ветки, увидим, что это самый первый, корневой (root) коммит, увидим количество попавших в индекс файлов (4 files changed) и суммарное количество новых строк (9 insertions), а также увидим перечень самих файлов и режим их создания. Про режим пока много думать не будем, это некое число, содержащее краткую информацию о том, как Git обрабатывает такой файл. В данном случае все файлы человекочитаемые.

А вот там, где информация о коммите, мы видим также странную комбинацию из букв и цифр (в данном случае «7cf3e89»). Это так называемый сокращённый хэш-код – часть полного хэш-кода. Если сокращённый хэш состоит из семи 16-ричных цифр, то полный аж из сорока (т.е. в реальности хэш коммита представляет собой 160-битное число).

Хэш-код может быть у любого объекта, в том числе у коммита, и используется для того, чтобы можно было дать каждому объекту некий уникальный номер, который обязательно (или практически обязательно) изменится, если изменится содержимое этого объекта. Само слово «hash» по-английски означает «фарш» или «мешанина», что как бы говорит нам о том, что значение хэша трудно предсказать.

Для чего нужен хэш-код? Чтобы, когда нужно получить информацию о состоянии файлов на момент какого-то выбранного коммита, этот коммит можно было легко и однозначно указать. Порядковый номер не подходит – если у нас имеется облачный репозиторий, наш локальный Git не может заранее знать, каким по порядку будет тот или иной коммит. Дата и время тоже – не исключено появление у разных сотрудников двух коммитов, созданных в один и тот же момент (да и время на компьютере может однажды сбиться). Поэтому единственный способ однозначно указать коммит – по его содержимому. Но по содержимому сравнивать коммиты на предмет того, это тот коммит, который нам нужен, или нет, неудобно – у нас может быть много файлов, которые различаются, скажем, на один байт где-нибудь на пятидесятом килобайте. К тому же, намного быстрее искать коммит среди прочих, когда свойство, которое проверяется на равенство, представляет собой число – тогда можно использовать алгоритмы и структуры

данных, значительно ускоряющие поиск. Так и пришли к идее хэш-кода – некоего очень длинного (чтобы уменьшить вероятность совпадения кода у двух разных объектов) числа, по которому можно быстро убедиться, что два объекта не равны.


под капотом

Как устроен хэш-код? Эта информация не требуется для освоения Git, так что этот раздел можно пропустить, но если есть желание разобраться, давайте немного углубимся в эту тему.

Для вычисления хэш-кода любого объекта по его содержимому используется так называемая хэш-функция. Выбор хэш-функции – задача непростая, которая требует знания теории криптографии. Хорошая, годная хэш-функция должна удовлетворять следующим критериям:

- Она должна вычисляться достаточно быстро (а иначе какой смысл вычислять и хранить её значение для объекта? Проще было бы сравнить просто по содержимому).
- Для одного и того же объекта, с неизменным содержимым, значение хэш-функции должно быть всегда одно и то же (что собственно проистекает из самого смысла хэш-функции – если у одного и того же содержимого в один момент один хэш, а в другой момент другой, то как мы по нему узнаем, что содержимое одинаково?).
- Заметим, что обратное в общем случае не только неверно, но и не может быть верным – для двух разных объектов вполне может получиться одинаковый хэш-код, просто по той причине, что различных хэш-кодов всё же меньше, чем вариантов содержимого (пусть их и 2^{160} штук). Вместе с тем, хорошая хэш-функция должна быть такой, чтобы вероятность такого совпадения (оно называется коллизией) была как можно меньше.

Для целей идентификации коммита этих свойств, в принципе, достаточно, но в иных целях (например, криптографических) к этим свойствам добавляют другие – например, то свойство, что по известному хэшу должно быть крайне сложно сконструировать объект, который бы имел такой код, и то свойство, что изменение даже одного бита в исходных данных должно приводить к очень сильно отличающемуся значению хэш-кода.

В принципе, не исключена ситуация, когда несколько коммитов содержат одинаковые изменения. Поэтому для уменьшения вероятности коллизии при расчёте хэш-кода коммита в качестве аргумента хэш-функции используется не одно лишь его содержимое, а его комбинация с метаданными (дата и время коммита, имя и e-mail пользователя, текст сообщения) и хэш-кодом предыдущего коммита (кроме корневого коммита).

Для расчёта хэш-кода Git использует алгоритм SHA-1 (Secure Hash Algorithm 1), в котором исходные данные разбиваются на блоки, которые многократно перетасовываются между собой. Для криптографических целей уже существуют и

более надёжные алгоритмы, но для целей различения коммитов между собой такого алгоритма вполне хватает.

Полученный хэш-код коммита настолько хорош, что во многих случаях хватает не полного набора из сорока 16-ричных цифр, а только первых 4-7 цифр. Поэтому зачастую коммиты задаются только сокращённым хэшем, и Git успешно ищет по нему нужный коммит. Если вдруг у нескольких коммитов в одном репозитории оказывается одинаковый сокращённый хэш-код (вероятность чего не очень велика, учитывая, что семь 16-ричных цифр уже даёт больше миллиона разных комбинаций, а даже небольшое изменение исходных данных приводит к лавинообразному изменению значения хэша), то мы просто используем в команде на одну-две цифры больше.

Теперь, когда мы снова, уже после коммита, запускаем git status, мы видим примерно следующий ответ от Git:

```
On branch master

nothing to commit, working tree clean
```

Таким образом, все файлы, которые ранее были неотслеженными, попали в репозиторий, и теперь они помечены как неизменённые – мы ведь после коммита их ещё не меняли.

Git ведёт как бы журнал сделанных изменений (точнее, сделанных коммитов). Посмотреть его можно по команде:

```
   git log
```

У этой команды есть ряд полезных ключей, но пока мы её запустим в чистом виде. Мы увидим примерно следующее:

```
$ git log

commit 7cf3e89dcdacb5329e0ccbd72c3741a080c17de8 (HEAD -> master)

Author: Гершман Антон Лейбович <agershman@corp.local>

Date:   Mon Dec 22 15:17:25 2025 +0300


    Начали работу над проектом
```

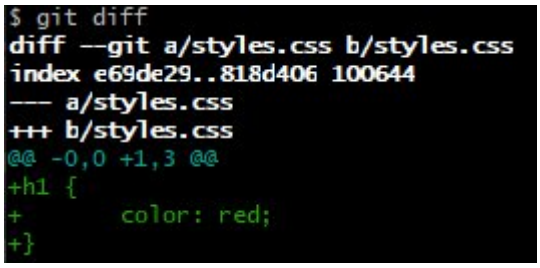
Мы видим полный хэш-код коммита, информацию о его авторе и времени его создания и сообщение коммита.

Перед тем, как делать коммит, можно получить сводку произведённых изменений. Добавьте какой-нибудь текст в файл, допустим, styles.css. Сохраните изменения и выполните команду

```
   git diff
```

(от английского difference – разница).

Вы увидите, в каких файлах были сделаны изменения, и какие именно. При этом добавленные в файл строки обозначаются значком «+», а удалённые – значком «-»:



Здесь в файл styles.css были добавлены строки

```
h1 {
        color: red;
}
```

При помощи этой команды можно сравнивать не только состояние рабочей папки с последним коммитом, но и любые два коммита между собой. Для этого нужно указать хэш-коды (можно краткие) в качестве параметров команды:

```
git diff 7cf3e89 a1b2c3d
```

Можно также указать не хэш-код коммита, а относительную ссылку на него:

```
git diff HEAD~1 HEAD
```

HEAD – это специальная ссылка, которая всегда означает тот коммит, в котором мы сейчас находимся. Ссылки вида HEAD~n указывают на коммит, находящийся на n пунктов раньше HEAD в журнале. Кроме HEAD, есть также другие виды ссылок – названия веток и теги, мы обсудим это позже.

Чтобы отменить сделанные изменения, также служит команда git reset. Только теперь, чтобы изменения не просто вышли из индекса (мы их туда пока и не добавляли), но и вообще были уничтожены, нужно добавить к ней флаг --hard. Будьте осторожны с этим флагом – будут уничтожены все изменения, которые вы внесли с момента последнего коммита.

Можно также исправить уже сделанный коммит – такое может пригодиться, если мы забыли добавить к индексу какой-то файл или, наоборот, добавили файл, который добавлять было не нужно, а также, если ввели неправильное сообщение коммита. Самый примитивный (но не самый лёгкий) способ – это сначала удалить коммит (но сохранить сделанные в нём изменения), а потом создать его заново, но уже правильно. Для первого шага мы опять же выполняем команду git reset, но указываем конкретный коммит, на который следует перейти. Нам надо на один коммит ранее последнего. Последний коммит для универсальности можно называть не только по хэшу, но и словом HEAD. Чтобы перейти на один коммит

ранее последнего, мы пишем в качестве названия коммита HEAD~1. Кроме того, нам надо, чтобы изменения, сделанные в последнем коммите, не пропали, а остались даже не в рабочей копии, а прямо в индексе, готовые к коммиту – для этого служит флаг soft:

```
git reset HEAD~1 --soft
```

После выполнения этой команды репозиторий переходит в то состояние, в котором он был прямо перед последней командой git commit. Теперь мы удаляем неправильный файл из индекса:

```
git rm --cached имяфайла
```

Если вы уже знакомы с консольными командами (командами терминала) для работы с файлами, возможно, вас насторожит команда rm (от английского «remove» – «удалить»). Но ничего страшного – в данном случае, с ключом cached, эта команда удалит файл только из индекса, оставив его в рабочей папке на диске без изменений. Кстати, в современных версиях Git есть более интуитивная команда, выполняющая те же действия (убирает файл из индекса, но оставляет его на диске):

```
git restore --staged имяфайла
```

Здесь же мы добавляем к индексу те файлы, которые забыли добавить (та же команда git add, которую мы уже знаем) и, наконец, снова выполняем коммит.

В принципе, этого достаточно, чтобы исправить последний коммит. Но есть также другой вариант: мы, не делая git reset, просто удаляем неверно добавленные файлы командой git rm --cached, добавляем забытые файлы командой git add, и опять вызываем команду git commit, но уже с флагом amend:

```
git commit --amend -m "сообщение"
```

Если мы хотим оставить прежнее сообщение, мы вместо флага –m и последующего текста можем использовать флаг --no-edit.

Слово «amend» буквально и означает «изменение», но фактически Git при этом не редактирует коммит, а создаёт новый от того же родителя, что и «исправляемый» коммит. Старый коммит остаётся в истории, но уже не привязанный к текущей ветке (про работу с ветками мы ещё поговорим немного позже).

ВАЖНО: исправлять коммит, будь то вариант с явным git reset или вариант с флагом amend, можно ТОЛЬКО если он ещё не ушёл в облако (отправку и получение коммитов мы рассмотрим чуть позже) или, если ушёл в облако, то только если этот коммит ещё не получен другими сотрудниками. В противном случае репозитории у разных сотрудников рассинхронизируются между собой, и приведение их к одному варианту может потребовать существенных трудозатрат.

Вопросы для самопроверки:

1. При помощи какой команды можно превратить папку проекта в локальный репозиторий?

   Варианты ответов:

   1.1. git create

   1.2. git start

   1.3. git init

   1.4. git repository --new

2. Что делает команда git commit -m?

   Варианты ответов:

   2.1. Выполняет коммит и выводит список сообщений ранее выполненных коммитов

   2.2. Выполняет коммит и открывает редактор для ввода сообщения коммита

   2.3. Выполняет коммит без сообщения

   2.4. Выполняет коммит с сообщением, которое указано в кавычках после ключа -m

3. Какой командой посмотреть изменения в рабочей папке по отношению к последнему коммиту?

   Варианты ответов:

   3.1. git changes

   3.2. git diff

   3.3. git compare

   3.4. git difference

§ 1.7. Отправка и получение изменений

Как мы уже знаем, есть два способа начала работы над новым проектом с использованием облачного репозитория Git. Первый – создать репозиторий в облаке и склонировать его себе – мы уже освоили. Но он не подойдёт для ситуации, когда у нас уже есть локальный репозиторий, и мы хотим его добавить в облако. Это может понадобиться, например, если мы сначала пользовались только теми возможностями, которые Git предоставляет локально – контролем версий самим по себе – а в облако решили добавить только потом, когда у нас появилось больше одного человека в команде. Или, например, мы пользовались одним облачным репозиторием, а в процессе работы оказалось необходимо перейти на другой (например, прежний больше недоступен или плата за его использование оказалась непосильной). В этом параграфе мы узнаем, как отправить в облако локальный репозиторий и как изменения, отправленные в облако, получить на локальной машине.

Итак, у нас с прошлого параграфа есть локальный репозиторий, в котором есть как минимум один коммит. Как теперь отправить всё это на облако?

Во-первых, нам нужен репозиторий в облаке. Как мы уже делали в параграфе про создание репозитория, зайдите на сервис и создайте там репозиторий. **ВНИМАНИЕ:** репозиторий создавайте пустой, без файлов по умолчанию (README.md и т.д.). После создания облачного репозитория откроется страница, с которой этот репозиторий можно клонировать.

Скопируйте строчку, которая используется для клонирования репозитория через SSH. Только теперь мы эту строчку будем использовать не для клонирования облачного репозитория на локальную машину, а для связи локального репозитория со свежесозданным облачным. Для этого в папке своего локального репозитория выполните команду:

```
git remote add origin git@сайт:итакдалее
```

Здесь origin – имя, под которым в локальном репозитории будет зарегистрирована ссылка на удалённый репозиторий (мы уже говорили об этом в параграфе, в котором рассматривали клонирование репозитория).

Теперь наш локальный репозиторий связан с удалённым репозиторием, так же, как если бы мы склонировали локальный репозиторий из удалённого.

Осталось собственно отправить наши коммиты (впрочем, пока он только один) на удалённый репозиторий. Для этого используем команду:

```
git push –u origin master
```

Флаг -u – это сокращение от --set-upstream (установить поток). Дело в том, что мало связать репозитории. Нужно ещё связать ветку локального репозитория с

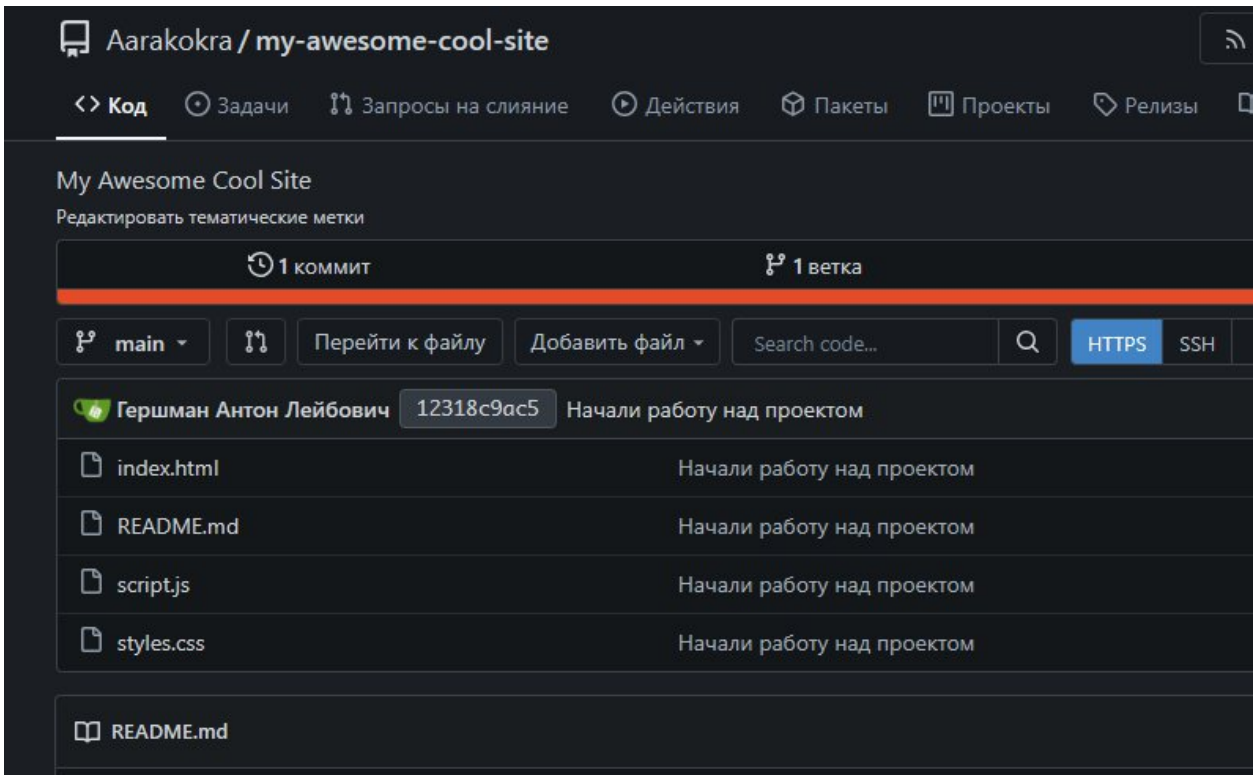
веткой в облачном репозитории. При клонировании это делается автоматически, но поскольку клонирования у нас не было, мы явным образом говорим при помощи флага –u, что для ветки, коммиты которой мы отправляем (это текущая ветка локального репозитория, в данном случае master) на облачный репозиторий, зарегистрированный под указанным именем (в данном случае origin), мы ищем в облачном репозитории ветку с указанным названием, т.е. master (а если её нет, создаём её).

Флаг –u достаточно использовать один раз для каждой ветки (при условии, что у нас один облачный репозиторий для данного локального репозитория, конечно). После этого связь устанавливается, и последующие вызовы команды git push производятся уже без этого ключа.

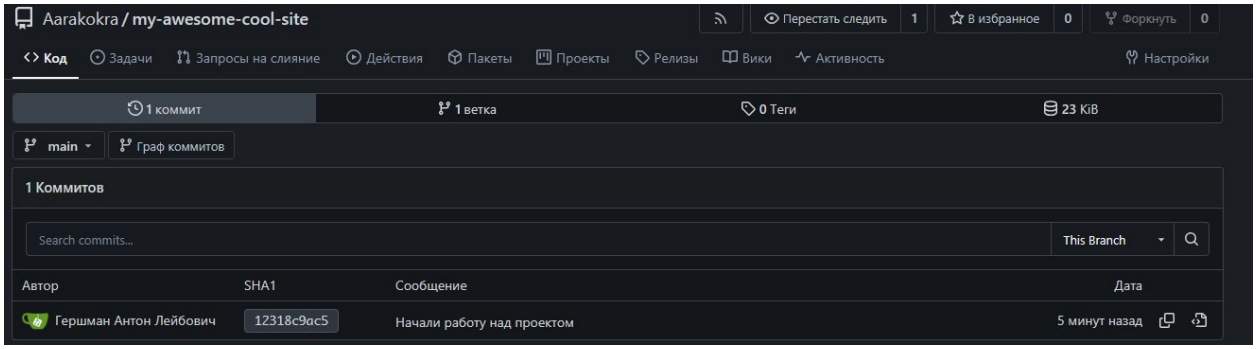
Если ваш закрытый ключ был защищён паролем (что желательно), то локальный Git попросит указать пароль. После этого локальный Git связывается с удалённым сервисом, проходит аутентификацию по SSH и отправляет накопленные коммиты в облако. На экране после этого появляется примерно следующая информация:

```
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (5/5), 502 bytes | 251.00 KiB/s, done.
Total 5 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To сайт:пользователь/репозиторий.git
 * [new branch]      master -> master
branch 'master' set up to track 'origin/master'.
```

Если теперь обновить страницу вашего репозитория на сайте (или зайти на неё заново, если вы её закрыли), вы увидите, что на странице появится информация о залитом коммите:



Если мы зайдём в пункт «Коммиты» (на скриншоте это значок часов с надписью «1 коммит», но в других облачных сервисах он может выглядеть по-другому и находиться в ином месте), то увидим список коммитов в облачном репозитории (пока там только один коммит – тот, который мы отправили в облако командой push):



Мы здесь видим имя ветки, сообщение коммита, его автора и его сокращённый хэш-код.

Допустим, ваш коллега склонировал себе этот репозиторий до того, как вы выполнили коммит. Как ему получить ваш коммит себе в локальный репозиторий?

Для этого он использует команду:

```
git pull
```

Тем, кто знает английский, запомнить эти команды легко: «push» означает «затолкать», а «pull» – «вытащить». Иногда и русскоязычные разработчики говорят «пушить» и «пуллить».

На самом деле, git pull представляет собой последовательное выполнение двух разных действий – fetch и merge либо fetch и rebase. Но подробнее на этом мы остановимся уже во второй части учебника.

Заметим, что если несколько коллег сделали коммиты локально и хотят выполнить push, то это сможет сделать только тот, кто успеет первым. Чтобы избежать возможных конфликтов между разными версиями файлов, Git следует правилу: пуш какой-либо ветки допускается делать только тогда, когда самый свежий коммит этой ветки, находящийся в облаке, уже есть в локальном Git того сотрудника, который выполняет пуш. Если вы попытаетесь сделать пуш, а в облаке уже есть более свежие коммиты, Git выдаст ошибку – примерно такую:

```
! [rejected] master -> master (fetch first)

error: failed to push some refs to и так далее
```

Таким образом, Git не даст командой git push удалить чужую работу. Нужно сначала сделать пулл, при возникновении конфликта устранить его (мы это ещё будем изучать), и только потом делать пуш.

Фактически, командная работа с облачным репозиторием во многом заключается в своевременных командах git pull и git push. Вместе с тем, Git предоставляет ряд инструментов для облегчения командной работы и соблюдения определённой иерархии ответственности. Всё это мы будем изучать в последующих параграфах.

Вопросы для самопроверки:

1. При помощи какой команды можно отправить локальный коммит в удалённый репозиторий?

   Варианты ответов:

   1.1. git pushing

   1.2. git push

   1.3. git send

   1.4. git upload

§ 1.8. Список игнорирования – файл .gitignore

Вы уже научились делать коммиты и отправлять их в облако командой push. Но что, если в рабочей папке лежит файл (с именем, скажем, .env) с паролем от базы данных? Стоит один раз утратить бдительность, и секрет окажется в облачном репозитории, а значит, любой, кто получит доступ к репозиторию, получит и пароль от базы. Давайте разберём, как этого избежать.

Если вручную отслеживать, какие файлы вы добавляете к коммиту, то вероятность ошибки, как бы мы внимательно ни работали, не равна нулю, а цена её может быть очень высока. Но ведь нам фактически надо просто раз за разом не допускать к индексации одни и те же файлы, это же задача не для интеллекта, а для автоматизации! И совершенно верно, эту задачу можно и нужно автоматизировать. Для этого служит список игнорирования.

Технически список игнорирования в Git устроен как лежащий в корневой папке репозитория файл .gitignore (обратите внимание на точку в начале имени). В этом файле мы просто перечисляем те файлы и папки, которые Git должен игнорировать при выполнении таких команд, как git add, git status, git diff и прочих. Таким образом, однажды правильно настроив список игнорирования, мы имеем все шансы навсегда избавить себя от риска случайно поделиться секретными данными. Кроме того, как мы уже упоминали ранее, в список игнорирования можно – и нужно – внести файлы, которые всё равно строятся автоматически. Во-первых, мы таким образом избавимся от засорения истории изменения репозитория многочисленными автоматическими перестроениями, а во-вторых, существенно сократим хранимый в репозитории объём данных. При создании приложений в виде исполняемых файлов (например, на C# или на Java) к генерируемым автоматически файлам относятся, например, создаваемые при компиляции проектов файлы с расширениями .exe, .dll, .obj, .class и т.д. При создании веб-интерфейсов на Angular к таковым относится содержимое папки node_modules. Наконец, в список игнорирования необходимо занести настройки и временные файлы – они у каждого участника команды могут быть свои, и если их изменения хранить в Git, коллеги будут просто мешать друг другу.

Итак, первое, что необходимо занести в список игнорирования – это файлы с секретными данными. Например, во многих видах проектов секретные настройки содержатся в файле .env (обратите внимание на точку в начале), от слова «environment» (в данном случае устоявшийся перевод – «окружение»). При разработке бэкенда на Java с прицелом на дальнейшую поставку через Docker также бывает, что секретные данные складывают в виде отдельных файлов в папку secrets в корневой папке проекта. Файлы .env можно встретить в проектах на NodeJS, Python, Golang, а также в проектах, использующих контейнеризацию через Docker. В них в формате «ключ=значение» могут храниться, в частности, пароли к базам данных, ключи доступа к сторонним платным API и т.д. Утечка таких данных может дорого обойтись.

Создайте в корне нашего проекта MyAwesomeCoolSite (или другого) файл .gitignore и откройте его в каком-нибудь чисто текстовом редакторе (какой-нибудь Word или Write не подойдёт – они работают не с чистым текстом, а с форматированным, и могут добавить в файл свою информацию, которую в редакторе вы не увидите, а для Git файл станет испорчен). Можете использовать Блокнот, Notepad++ или, если вы находитесь в Git Bash, уже известные нам Vim или Nano.

Чтобы внести в список игнорирования файл или целую папку, просто укажите их в файле .gitignore:

```
.env

secrets
```

Проверим, что список игнорирования работает. Выйдите из редактора с сохранением изменений, создайте файл .env с каким-нибудь содержимым (например, password=123) и папку secrets с каким-нибудь файлом (скажем, файл adminpassword с содержимым qwe). Запустите команду git status. Она должна показать только одно изменение – появление, собственно, файла .gitignore (сам-то список игнорирования добавлять в коммит можно и даже нужно, чтобы он работал у всех участников команды). Ни файл .env, ни папка secrets, ни содержимое этой папки появиться не должны. Они теперь игнорируются, и случайно добавить их в коммит мы уже не сможем.

При разработке на Java файлы обычно распределяются по многим папкам, в зависимости от их назначения. Допустим, у нас код бэкенда на Java находится в папке src, а внутри этой папки есть папка secrets, содержащая код для работы с секретными данными (структура осознанно упрощена – мы сейчас изучаем Git, а не Java). Создайте в корне репозитория папку src, внутри неё папку secrets, а внутри этой папки файл secretservice.java с каким-нибудь содержимым. То есть, идея в том, что имя папки, которая должна отслеживаться, случайно совпало с именем папки, которая должна игнорироваться. Вернитесь в корень репозитория и выполните команду git status. Мы увидим, что файл secretservice.java, равно как и содержащая его папка, также игнорируется. Как же быть?

Для того, чтобы игнорировать не все папки secrets, а только ту папку secrets, которая находится в корне репозитория, можно явным образом указать, что папка находится в корне. Git при анализе файла .gitignore делает вид, будто корень репозитория является корнем файловой системы вообще, поэтому нам достаточно просто вместо secrets написать /secrets:

```
.env

/secrets
```

После того, как вы сохраните отредактированный .gitignore, опять выполните git status. Вы увидите, что теперь игнорируется только та папка secrets, в которой мы действительно сложили секретные данные. Если мы выполним команду

```
git add .
```

и снова выполним git status, то мы увидим, что файл secretservice.java из папки src/secrets присутствует в индексе.

Итак, мы добавили секретные данные в список игнорирования. Следующее, что мы совершенно точно не хотим видеть в коммитах – это автоматически создаваемые файлы. Например, раз уж у нас есть бэкенд на Java (ну то есть по факту его нет, но мы делаем вид, будто есть), то при его перестроении будут появляться исполняемые файлы в папках bin и/или build. Добавим их в файл .gitignore:

```
.env

/secrets

bin

build
```

Всё хорошо, но как-то неаккуратно. Файл .gitignore у нас выглядит несколько похоже на свалку – если заранее не знать, то не очень понятно, где заканчивается перечисление секретных данных, а где начинается перечисление автоматически создаваемых файлов и папок. Вот если бы, как в коде, можно было вставлять комментарии...

А кто говорит, что нельзя? Мы можем в файл .gitignore вставлять комментарии. Строчка комментария должна начинаться с символа #:

```
# Секретные данные

.env

/secrets


# Автоматически создаваемые файлы и папки

bin
```

```
build
```

Стало намного аккуратнее.

Наконец, нам нужно добавить в список игнорирования временные и настроечные файлы, создаваемые средой разработки (IDE). Например, если вы пишете сайт с использованием редактора Visual Studio Code, такие файлы создаются в папке .vscode:

```
# Секретные данные
.env
/secrets

# Автоматически создаваемые файлы и папки
bin
build

# IDE
.vscode
```

Однако бывает так, что некоторые файлы, создаваемые IDE, мы бы хотели иметь одинаковыми во всей команде. К таковым могут относиться файл настроек проекта settings.json, файл настройки команды запуска отладчика launch.json и файл настроек специфичных для проекта расширений extensions.json. Если мы хотим эти файлы исключить из списка игнорирования, несмотря на то, что вся остальная папка .vscode добавлена, мы можем после указания этой папки явным образом их перечислить, предваряя каждый символом «!» – этот символ укажет Git, что помеченный таким образом файл игнорировать не следует:

```
# Секретные данные
.env
/secrets

# Автоматически создаваемые файлы и папки
bin
build

# IDE
.vscode
```

```
!.vscode/settings.json

!.vscode/extensions.json

!.vscode/launch.json
```

Добавьте изменения в файл .gitignore в индекс командой git add .gitignore (или git add .), а затем выполните коммит с сообщением, например, «Настроили .gitignore» или «Добавили список игнорирования» и отправьте его в облако.

Файлы, специфичные конкретно для вашей и только вашей среды разработки (например, .idea/ для IntelliJ или .DS_Store для macOS), лучше игнорировать не в проекте, а глобально для всей вашей машины. Это делается командой:

```
git config --global core.excludesfile ~/.gitignore_global
```

После этого создайте файл ~/.gitignore_global и добавьте туда правила, которые должны работать для всех ваших проектов на этом компьютере.

Следует заметить, что Git понимает использование стандартной нотации шаблонных символов пути к файлам в .gitignore.

Например, при запуске отладчика наше приложение создаёт множество логов с расширением .log. Мы можем все их игнорировать, указав при помощи звёздочки, что имя файла может быть любым, но заканчиваться должно на .log:

```
*.log
```

Если нам нужно игнорировать логи только в папке repositories, можно написать так:

```
repositories/*.log
```

а если в подпапке папки repositories, то так:

```
repositories/*/*.log
```

Но звёздочка «не переходит через слеш» – так мы укажем, что нужно игнорировать логи в непосредственной подпапке папки repositories. Если нужно игнорировать логи во всех подпапках всех подпапок папки repositories, на любую глубину, используется двузвёздочная нотация:

```
repositories/**/*.log
```

А что будет, если мы уже ранее скоммиттили файл, а теперь хотим его игнорировать?

Увы, но простое добавление файла в список игнорирования не удалит его из уже выполненных коммитов. Он останется в истории. И даже если вы его потом удалите, это удаление не попадёт в коммит, если файл уже находится в списке игнорирования – ведь Git больше не отслеживает его изменения. Поэтому

правильный порядок действий в такой ситуации: сначала скопировать этот файл в какое-нибудь резервное место вне репозитория (если он, конечно, не относится к числу автоматически создаваемых), затем удалить его из репозитория, эти изменения проиндексировать и скоммиттить, и вот уже после этого добавить файл в список игнорирования, вернуть его из резервного места (если он был скопирован) и выполнить ещё один коммит для фиксации изменений в .gitignore. Если коммит с этим файлом был только один, есть способ проще – выполнить amend и rm --cached. Но в общем случае придётся идти длинным путём.

Когда мы создаём репозиторий не из локальной папки, а на сайте (тот способ, который мы изучали первым), то многие облачные сервисы предлагают сразу создать файл .gitignore исходя из выбранного нами типа проекта. В GitHub и в Gitea это предлагается явным образом, в GitLab зависит от настроек, в МосХабе (который, помним, по факту клон GitLab) эта опция доступна, если при создании проекта выбрать режим «Создать из шаблона». В зависимости от выбранного шаблона будет сразу создана та или иная типовая структура файлов и папок, и в том числе будет создан подходящий для конкретного случая файл .gitignore.

Вопросы для самопроверки:

1. Для чего служит список игнорирования?

   Варианты ответов:

   1.1. Чтобы удалять ненужные файлы из репозитория.

   1.2. Чтобы автоматически исключать некоторые файлы из отслеживания.

   1.3. Чтобы шифровать секретную информацию перед добавлением в репозиторий.

   1.4. Чтобы сжимать большие файлы перед записью в облако.

2. Что произойдёт, если мы добавим в список игнорирования ранее закоммиченный файл?

   Варианты ответов:

   2.1. Файл автоматически удалится из репозитория.

   2.2. Файл останется в репозитории, но его последующие изменения, включая удаление, будут проигнорированы.

   2.3. Git выдаст сообщение об ошибке, и потребуется разрешение конфликта вручную.

2.4. Git продолжит отслеживать изменения в файле, но мы не будем видеть эти изменения при просмотре коммита.

3. Как можно добавить в список игнорирования папку, но исключить из списка игнорирования конкретный файл в этой папке?

   Варианты ответов:

   3.1. Записать в .gitignore эту папку, затем в другой строке указать этот файл вместе с папкой, но начать эту строку с восклицательного знака:

   ```
   папка
   !папка/файл
   ```

   3.2. Записать в .gitignore эту папку, затем в другой строке указать этот файл, но начать строку с символа комментария:

   ```
   папка
   #файл
   ```

   3.3. Записать в .gitignore эту папку, затем в другой строке указать этот файл, но начать строку с символа минуса:

   ```
   папка
   -файл
   ```

   3.4. Записать в .gitignore эту папку, а после названия папки в квадратных скобках указать этот файл:

   ```
   папка[файл]
   ```

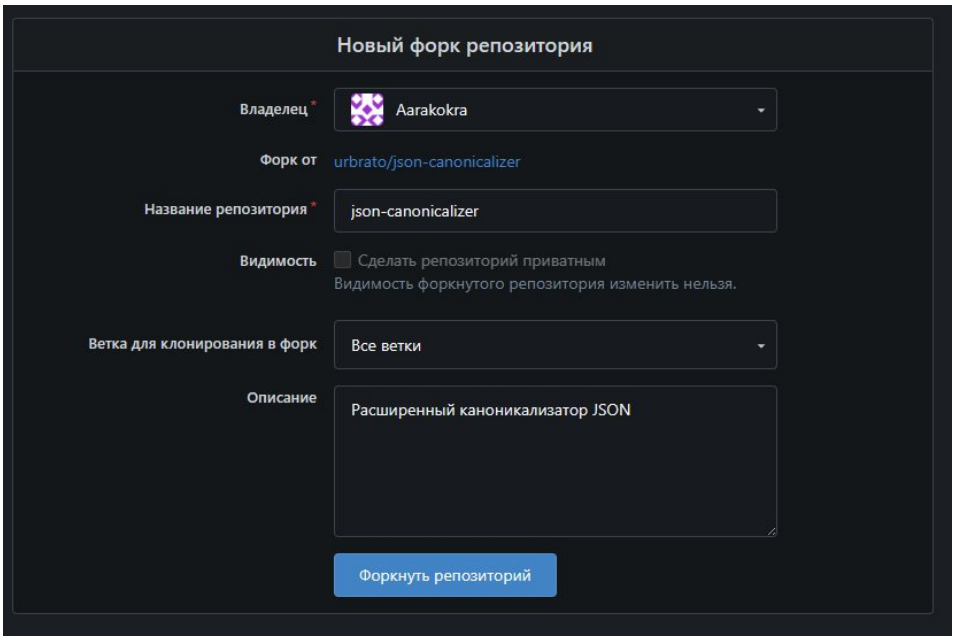
§ 1.9. Работа с чужим публичным репозиторием

Итак, вы уже научились создавать репозиторий локально и облачно, делать коммиты, а также заливать локальные коммиты в облако и утягивать коммиты из облака к себе. До сих пор это относилось только к тем репозиториям, которые вы создали самостоятельно. Однако, мы уже знаем, что облачные сервисы содержат также и публичные репозитории. Как работать с ними? Примерно так же, но есть нюансы.

Прежде всего, далеко не всегда у вас будет право что-то записать в чужой репозиторий. Если вы попытаетесь запушить коммит, Git выдаст ошибку. Но для себя вы всё же можете внести изменения в чужой проект. Для этого вам нужно сделать свою копию чужого публичного репозитория. Этот процесс по-английски называется «forking», от слова «fork» – «вилка». Русскоязычные разработчики тоже часто говорят «форкнуть», а созданную таким образом свою собственную копию чужого публичного репозитория называют форком этого репозитория.

Важно понимать разницу между командами fork и clone. Команда fork создаёт копию выбранного репозитория на сервере. У вас появляется свой репозиторий, которым управляете вы и который в принципе не обязан в дальнейшем как-то взаимодействовать с тем репозиторием, с которого он скопирован. Команда clone же скачивает облачный репозиторий на локальную машину и предназначена для того, чтобы в дальнейшем работать над этим же кодом.

Как правило, в веб-интерфейсе облачного сервиса на странице репозитория присутствует кнопка «Fork», «Форкнуть» или «Форк». После нажатия на эту кнопку откроется форма, похожая на форму создания нового репозитория, но с меньшим количеством настроек:



Точный набор настроек зависит от сервиса. Как правило, можно выбрать новое название для форка и ввести новое описание. В некоторых сервисах можно настроить уровень видимости (МосХаб), в некоторых указать, владельцем будет считаться ваш личный аккаунт или аккаунт организации, в которой вы состоите (GitHub). Также в некоторых сервисах можно выбрать, копировать весь репозиторий или только ветку по умолчанию. Если такая настройка в вашем сервисе присутствует, возможно, имеет смысл её включить, чтобы ветки, созданные для работы над теми или иными новшествами, не удалённые на настоящий момент, не занимали места (подробнее тема веток будет раскрыта позднее).

После того, как форк создан, его можно клонировать к себе на машину, как и любой собственный репозиторий.

Важно понимать, что форк, если ничего не предпринимать специально, больше не привязан к проекту, из которого он скопирован. Если мы склонируем репозиторий и затем выполним команду

```
git remote –v
```

то мы увидим, что в локальном репозитории ссылки на удалённый репозиторий ведут только на форк (и по умолчанию называются origin). Однако обычно мы хотим иметь возможность получать в свой форк также и все изменения, которые автор внёс в оригинал репозитория.

По счастью, Git позволяет нам связать локальный репозиторий более чем с одним удалённым репозиторием. Для этого выполним команду:

```
git remote add имя_ссылки_на_оригинал ссылка_на_оригинал
```

Здесь имя ссылки на оригинал мы выбираем произвольно. Есть соглашение, по которому для таких случаев используют имя «upstream», равно как для основной ссылки для push и pull используют имя «origin». Сама же ссылка на оригинал – это та ссылка, которую мы бы использовали, если бы хотели склонировать оригинал.

Теперь мы можем при помощи команды git pull получать свежие коммиты из нашего форка, а при помощи git pull имя_ссылки (например, git pull upstream) – свежие коммиты из оригинала. Не забывайте после обновления своего локального репозитория из оригинала (git pull upstream) обновлять форк в облаке (git push). Конечно, может возникнуть ситуация, когда коммиты из обоих источников касаются одного и того же места в одном и том же файле. Тогда возникнет конфликт коммитов, и его нужно будет устранить вручную (как это делается, мы будем рассматривать позже).

Не очень часто, но, тем не менее, бывает, что нам нужно не добавить новый удалённый репозиторий, а сменить имеющуюся ссылку (чаще всего это касается origin). Наиболее вероятные сценарии, когда это требуется:

- мы общались со своим основным удалённым репозиторием по HTTPS, но администрация сервиса ограничила или запретила эту возможность, и мы должны перейти на SSH;
- сервис удалённых репозиториев, которым мы пользовались, прекратил работу (был заблокирован, кончилось финансирование, ликвидировано ответственное за поддержание работоспособности сервиса юридическое лицо, и т.д., и т.п.), и нам надо мигрировать нашу работу на другой сервис;
- проект, который был задуман, как личный, и находился в личном репозитории, заинтересовал фирму, в которой вы работаете, и вы мигрируете его в корпоративный репозиторий (или наоборот, корпоративный проект больше не нужен, но вы не хотите его потерять и мигрируете в персональный);
- (реже, но тоже случается) мы банально допустили опечатку в url, когда выполняли команду git clone или git remote add.

Во всех этих случаях нам необходимо оставить имя origin, но сохранить под ним другой url. С этой целью используется команда

```
git remote set-url имя ссылка
```

Мы можем даже решить несколько задач (смена протокола и смена сервиса) одной командой Например, мы от имени корпоративного пользователя «acme» мигрируем репозиторий «superpupergame» со ставшего недоступным вследствие фатальной ошибки, дошедшей до продуктивной среды, МосХаба, с которым мы общались по SSH, на наш персональный репозиторий git.acme.ru (воспользуемся старинной традицией, по которой в качестве имярека для произвольно взятого названия организации, домена и т.п. используется название Acme), с которым мы общаемся по HTTPS. Для этого, в предположении, что у нас локально имеется последняя версия репозитория, нам достаточно выполнить команду:

```
git remote set-url origin https://git.acme.ru/acme/superpupergame.git
```

После этой операции весьма желательно (можно даже сказать, необходимо) выполнить две проверки.

Во-первых, нужно убедиться, что само по себе изменение ссылки прошло корректно. Для этого выполним уже известную команду

```
git remote -v
```

Мы увидим, что вывод

```
origin git@hub.mos.ru:acme/superpupergame.git (fetch)
origin git@hub.mos.ru:acme/superpupergame.git (push)
```

сменился на

```
origin https://git.acme.ru/acme/superpupergame.git (fetch)
```

```
origin https://git.acme.ru/acme/superpupergame.git (push)
```

А вторая проверка, которую обязательно нужно провести – убедиться, что работа с новым репозиторием возможна. Для этого просто делаем git fetch (обратите внимание, что сам удалённый репозиторий должен на этот момент существовать). Это нужно не только для проверки работоспособности ссылки, но и для того, чтобы обновить информацию об удалённых ветках. Если всё прошло без инцидентов – мы можем завершить миграцию, сделав пуш. Удалённый репозиторий для этого должен не просто существовать, но, более того, быть пустым (в нём не должно быть ещё ни одного коммита), в противном случае возникнет расхождение истории и пуш не будет выполнен. Если нам прежнее содержимое удалённого репозитория не нужно, мы выполняем пуш ещё раз, но теперь с флагом force, который в данном случае фактически просто заменит старое содержимое на новое:

```
git push --force
```

Обратите внимание – флаг force заставляет Git работать по очень опасному сценарию, всё содержимое удалённого репозитория, которое отсутствует у вас локально, может быть (и, скорее всего, будет) безвозвратно утеряно. Используйте его только тогда, когда вы абсолютно точно уверены в том, что прежнее содержимое не пригодится (например, вы, создавая удалённый репозиторий, случайно создали его не пустым, а с файлом README.md). Чуть более безопасно использовать вместо этого другой флаг:

```
git push --force-with-lease
```

Это также уничтожит прежнее содержимое удалённого репозитория, но только в том случае, если его автор вы. При наличии коммитов, выполненных другими пользователями, команда завершится с ошибкой.

Команда set-url не меняет историю коммитов и содержимое файлов, но следует уделить особое внимание тому, чтобы при первом пуше в новый удалённый репозиторий не потерять информацию, которая там содержалась. А лучше – мигрируйте всегда в чистый пустой репозиторий.

Кроме публичных репозиториев, открытых для клонирования и получения свежих коммитов, но закрытых для пуша туда своих коммитов, есть также и репозитории, куда вы можете вносить свои изменения – репозитории открытого исходного кода, или опенсорсные репозитории (open source).

Для того, чтобы вносить свой вклад в развитие проекта с открытым исходным кодом, обычно требуется следовать некоторым правилам, так сказать, опенсорсного этикета. Вместе с тем, участие в такой деятельности – это отличный способ получить опыт и добавить что-то в своё портфолио.

Одно из правил этикета участников проекта с открытым исходным кодом гласит: прежде чем вносить изменения в чужой open-source проект, первым делом найдите в корне репозитория файл CONTRIBUTING.md (Руководство для участников). Там

авторы проекта описывают свои правила: как оформлять коммиты, как запускать тесты и куда именно отправлять свои предложения. Если такой файл есть, пожалуйста, следуйте прописанным в нём рекомендациям.

Вопросы для самопроверки:

1. Для чего делается форк чужого репозитория?

   Варианты ответов:

   1.1. Для ускорения работы с репозиторием.

   1.2. Для того, чтобы иметь возможность вносить правки в код, если в чужом репозитории у нас таких прав нет.

   1.3. Для переименования репозитория.

   1.4. Для того, чтобы заменить чужой репозиторий своим.

2. После того, как мы склонировали наш форк чужого репозитория на локальную машину, какой командой можно добавить чужой репозиторий в качестве дополнительного источника изменений?

   Варианты ответов:

   2.1. **`git add`** *`имя ссылка`*

   2.2. **`git clone`** *`имя ссылка`*

   2.3. **`git remote add`** *`имя ссылка`*

   2.4. **`git push`** *`имя ссылка`*

3. Как обычно по соглашению называют ссылку на оригинал форка?

   Варианты ответов:

   3.1. origin

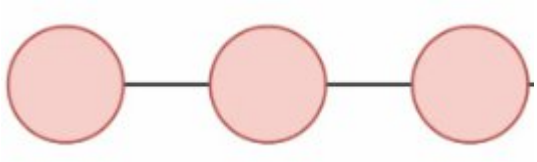
   3.2. main

   3.3. source

   3.4. upstream

§ 1.10. Ветки

Итак, вы уже умеете делать коммиты и отправлять их в облачный репозиторий. Однако, до сих пор история коммитов была, можно сказать, линейной – если нарисовать все коммиты в виде кружочков и от каждого коммита нарисовать стрелочку к предыдущему коммиту (или к начальному состоянию репозитория), можно их все вытянуть в одну цепочку. Эта цепочка – так называемая ветка по умолчанию или главная ветка.



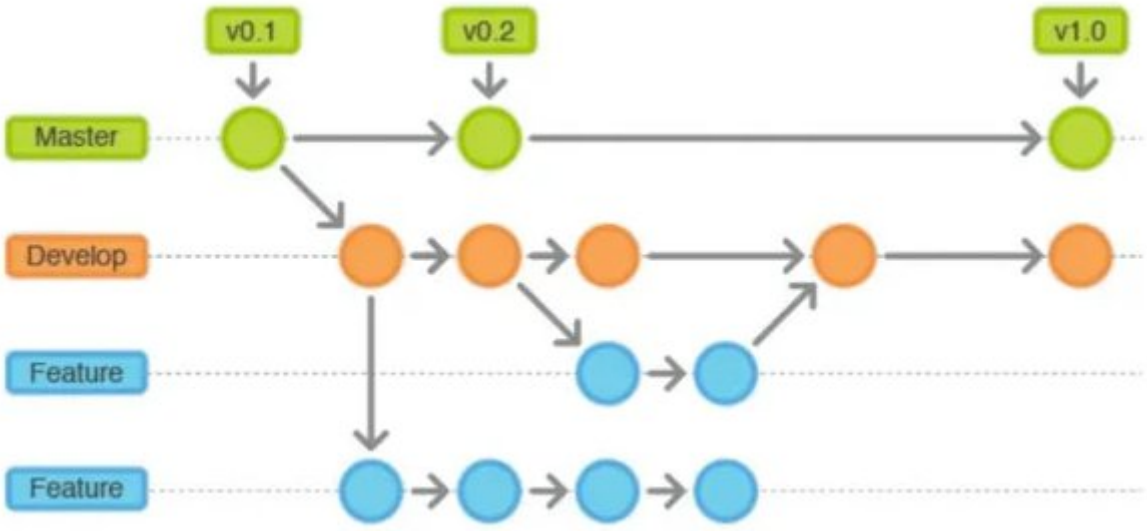
Коммиты в одной ветке.
Здесь это ветка по умолчанию – других веток нет.

При работе с небольшими проектами и небольшими силами этого достаточно, но когда ваш проект становится достаточно крупным, или когда вы начинаете работать с опенсорсными проектами, очень важно научиться работе с ветками кода: создавать их, сливать вместе и удалять. А для этого, разумеется, нужно сначала понять, что это вообще такое.

Допустим, команда разработчиков работает над одним проектом. Кто-то из разработчиков реализует одну новую функцию, кто-то другую, кто-то исправляет найденную ошибку. Все они делают коммиты в репозиторий. Если бы не было механизма веток, было бы, во-первых, сложно понять, какой коммит к какой функции или к какому исправлению относится, а во-вторых, разработчики мешали бы друг другу – ведь мы уже знаем, что пуш какой-либо ветки допускается делать только тогда, когда самый свежий коммит этой ветки, находящийся в облаке, уже есть в локальном Git того сотрудника, который выполняет пуш, а если это не так, нужно сначала сделать пулл. Наконец, если мы работаем с опенсорсным проектом (а иногда и в пределах своего предприятия, если у нас ещё немного опыта в используемых технологиях или в доработке данной конкретной системы), бывает, что мы в принципе не имеем права делать пуш в основную ветку, чтобы случайно не выложить негодный код. Во всех этих случаях нас выручит механизм веток.

Но что же такое ветки?

Основная ветка – это как бы история развития нашего продукта. Остальные ветки – это как бы альтернативная история, другие пути развития. Технически, если мы, опять же, нарисуем коммиты в виде кружочков и нарисуем стрелки от каждого коммита к предыдущему, мы можем увидеть, что у нескольких коммитов есть общий предыдущий коммит, то есть, при движении по истории развития продукта в какой-то момент история раздвоилась. Вот такое раздвоение и есть создание новой ветки.



Коммиты в разных ветках.
Здесь стрелки направлены хронологически – от предыдущих коммитов к последующим.

При работе в одной ветке мы никак не влияем на другие ветки. Мы можем вносить любые изменения, экспериментировать, и основная ветка продукта никак от этого не изменится. Наконец, когда мы считаем, что работа над функцией или над исправлением ошибки завершена, мы можем согласовать сделанные изменения с руководителем проекта (если это практикуется) и одной командой внести все сделанные нами изменения в основную ветку. Если представить основную ветку проекта в виде дороги, по которой идёт его развитие, то ветку можно представить, как отходящую от этой дороги тропинку, которая произвольным образом идёт то туда, то сюда, а в конце концов снова вливается в главную дорогу. После того, как все изменения, сделанные в «боковой» ветке, попали в главную ветку, «боковая» ветка больше не нужна, и её можно удалить. При этом коммиты, которые были в ней сделаны, не пропадут. Если же мы пришли к выводу, что всё, что делалось в ветке (кроме основной), неактуально, мы можем просто удалить ветку, не внося её коммиты в основную ветку. Вот тогда все её коммиты, не входящие в основную ветку, пропадут.


под капотом

Почему так? Потому что технически, «под капотом», веток как отдельных «тропинок» из ответвляющихся от основной ветки цепочек коммитов, не существует, коммиты не хранят информацию о том, что они относятся к какой-то там ветке. Коммит хранит информацию о себе (что естественно) и о том, какой коммит был для него предыдущим. Информация о том, какой коммит последний, хранится в некоторой, скажем так, ссылке (по факту, эта ссылка представляет собой хэш-код коммита). А ветки хранятся просто в виде ссылок на последний коммит каждой ветки. Зная последний коммит той или иной ветки, мы можем, следуя по цепочке ссылок на предыдущие коммиты, восстановить всю цепочку вплоть до точки, где ветка отпочковалась от другой ветки (кстати, не обязательно от основной), и далее вплоть до начального состояния репозитория. После того, как последний коммит ветки вошёл в состав основной ветки, и далее «пожизненно», мы можем попасть в

этот коммит и во все предыдущие для него коммиты, следуя от последнего коммита основной ветки по стрелочкам. Таким образом, отдельная ссылка на него (которая, «под капотом», и представляет собой ветку), действительно, больше не нужна. Если же последний коммит ветки не попал в основную ветку, то после удаления ссылки на него к нему больше нельзя пройти от какой-нибудь из существующих ссылок, и таким образом, и последний коммит, и все коммиты этой ветки, которые не являются предыдущими для каких-то других коммитов, пропадут – точнее, станут недостижимыми. Они не удалятся мгновенно из истории (Git их будет хранить примерно месяц, на случай, если произошла ошибка – в таком случае, её можно будет исправить путём довольно сложных манипуляций, рассмотрение которых выходит за рамки нашего учебника – но для обычной работы они будут потеряны, так как на них больше не указывает ни одна ветка).

Давайте начнём осваивать работу с ветками на практике. Практиковаться мы будем на репозитории MyAwesomeCoolSite, созданном вами ранее.

Чтобы посмотреть, какие ветки уже есть в репозитории, выполним команду:

```
git branch
```

Слово «branch» по-английски и означает «ветка».

Мы увидим примерно следующее:

```
* master
```

Разные версии Git и разные облачные сервисы могут именовать основную ветку по-разному, но обычно это либо master, либо main. Других веток мы не создавали, поэтому только одно это название мы и видим. Оно помечено звёздочкой в знак того, что мы сейчас «находимся» в этой ветке. Это означает, что ссылка на последний коммит будет указывать на последний коммит именно этой ветки, и что все новые коммиты будут попадать именно в неё.

НА ЗАМЕТКУ:

Строго говоря, исторически первым названием ветки по умолчанию было именно master (в переводе – ведущая, основная). Однако, с появлением общественного движения BLM стали поступать жалобы на то, что это название напоминает о временах рабовладения (поскольку это слово также переводится и как «господин»), и многие сервисы поменяли название по умолчанию с master на main (главная). Начиная с версии Git 2.28, вы можете настроить Git так, чтобы он автоматически создавал ветку «main» вместо «master» при каждом вызове «git init». Для этого достаточно один раз выполнить команду:

```
git config --global init.defaultBranch main
```

Чтобы понять, что такое ветки, лучше всего начать работать с ветками. Создадим новую ветку. Для этого также служит команда git branch, но теперь после неё через пробел нужно указать название ветки, которую мы хотим создать. Допустим, мы хотим добавить к имеющимся у нас в нашем репозитории файлам (а это у нас README.md, index.html, styles.css и script.js) описание того, для чего каждый файл служит. Например, мы не хотим добавлять это описание к основному файлу описания README.md, чтобы не засорять этой информацией главную страничку репозитория, а вместо этого хотим сделать отдельное описание в том же формате Markdown – пусть это будет файл filelist.md. Мы сейчас учимся, поэтому, хотя для такого небольшого инструмента ветка в принципе не очень-то и нужна, мы сделаем вид, будто это очень большое и серьёзное изменение, которое теоретически может всё поломать, так что заведём для этого новую ветку.

Как выбрать имя для ветки? На этот счёт в разных компаниях и даже в разных рабочих группах достаточно крупных компаний есть разные соглашения. Где-то называют ветку по номеру задачи в системе отслеживания задач, где-то называют ветку по содержанию того, что в ней предполагается делать, где-то к содержанию добавляют какую-нибудь условную последовательность букв для обозначения, мы делаем какую-то новую «фичу» («feature») или исправляем обнаруженную ошибку (например, начинают имя ветки с «feature/», если это новая «фича», или с «fix/», если это исправление ошибки). Система отслеживания задач на большинстве сервисов сейчас есть, но мы её не использовали, так что номера задачи у нас нет. Поэтому назовём ветку просто «feature/filelist»:

```
git branch feature/filelist
```

Git создал ветку с указанным именем. Мы это можем проверить, снова выполнив команду git branch. Мы увидим примерно такой ответ:

```
  feature/filelist
* master
```

Мы видим, что в списке веток появилась ветка с именем «feature/filelist», как мы и указали, но текущая ветка всё равно сейчас master. Это важно – команда git branch с именем ветки создаёт ветку с таким именем, но не выполняет переход на неё.

Кстати, использование слэша («/») в имени ветки – это не просто стиль. Многие графические клиенты (например, SourceTree, GitKraken или панель Git в составе Android Studio) автоматически сворачивают такие ветки в удобные папки feature/, fix/, docs/, что сильно упрощает навигацию, когда веток становятся десятки.

Как нам теперь переключиться на новую ветку? До версии Git 2.23 для этого служила команда **git checkout** *имя ветки*. Но с течением времени команда git

checkout получила очень много разных опций (едва ли не больше, чем значений у этого слова в английском языке), и, чтобы не путаться с их использованием, решили продублировать часть из них отдельными командами. Так, начиная с версии Git 2.23, появилась команда **git switch** *имя ветки*, которая занимается именно переключением веток (слово «switch» и означает «переключить»). Чтобы уменьшить вероятность перепутать разные режимы команды git checkout, для переключения веток настоятельно рекомендуется использовать именно новую команду:

```
git switch feature/filelist
```

Если теперь при помощи команды git branch посмотреть на список веток, мы увидим, что звёздочка перешла к нашей новой ветке.

А нельзя ли, когда мы создаём новую ветку, сразу же перейти на неё? Можно. Для этого можно использовать опять же хоть команду git checkout, хоть команду git switch. Чтобы переключиться при помощи этих команд на ветку, которой ещё не существует (точнее, конечно же, сначала создать её, а потом переключиться на свежесозданную ветку), нужно для команды checkout использовать флаг -b (от слова «branch» – «ветка»), а для команды switch – флаг -c (от слова «create» – «создать»):

```
git checkout –b feature/filelist
```

или

```
git switch –c feature/filelist
```

Итак, мы создали новую ветку, и что дальше?

Свежесозданная ветка практически ничем не отличается от той ветки, где мы были, когда мы её создали. Там те же самые файлы с тем же самым содержимым. Чтобы убедиться в этом, выполним команду git status. Мы увидим что-то такое:

```
On branch feature/filelist

nothing to commit, working tree clean
```

В принципе, это логично – создав новую ветку, мы, фактически, просто создали новую ссылку на тот же последний коммит, только под другим названием.

Но теперь, когда мы создали новую ветку и, главное, перешли в неё, мы можем добавлять в эту ветку новые файлы и менять содержимое существующих файлов, а в главной ветке при этом ничего не изменится. Когда мы будем коммитить изменения, эти коммиты появятся в ветке, в которой мы находимся, но не в ветке master.

Добавьте в папку проекта любым доступным вам способом файл filelist.md и напишите туда какое-нибудь описание имеющихся файлов. После того, как вы сохраните файл и снова выполните команду git status, вы увидите, что filelist.md появился в списке неотслеженных (untracked) изменений.

Закоммитьте сделанные изменения. Это нужно будет сделать в два этапа: сначала добавьте файл к индексу командой git add, а затем закоммитьте с сообщением «feat.: Добавлен список файлов». Здесь мы начали сообщение с «feat.:», чтобы, глядя потом на историю коммитов, можно было быстро определить, в каких из них мы добавляем новые «фичи».

Залейте коммит на облако при помощи команды

```
git push –u origin feature/filelist
```

Почему мы опять использовали флаг -u? Потому что мы заливаем новую ветку, которой ещё нет в облаке, а значит, нужно установить поток, т.е., создать в облаке ветку feature/filelist (если её ещё нет) и связать с ней ветку feature/filelist локального репозитория. Последующие пуши в эту ветку можно будет делать без данного флага.

Если мы теперь переключимся всё той же командой git switch на ветку master, мы увидим, что наш новый файл исчез! Но не беда – когда мы переключимся обратно, файл снова появится. Таким образом, у нашего репозитория теперь есть две ветки, и не только содержимое файлов, но даже их количество в них разное.

Вопросы для самопроверки:

1. Какая из этих задач решается созданием новой ветки?

   Варианты ответов:

   1.1. Удаление основной ветки (master либо main).

   1.2. Внесение изменений, которые не затрагивают основную ветку.

   1.3. Рассылка изменений другим пользователям репозитория.

   1.4. Изменение версии Git на своей рабочей машине.

2. Что означает звёздочка возле имени ветки в выводе команды git branch?

   Варианты ответов:

   2.1. Эта ветка помечена на удаление.

   2.2. Это основная ветка репозитория.

   2.3. Эта ветка синхронизирована с облачным сервисом.

   2.4. В этой ветке мы сейчас работаем.

3. Что делает следующая команда?
   `git push –u origin some_branch`

Варианты ответов:

3.1. Удаляет ветку some_branch из облачного репозитория.

3.2. Создаёт в облачном репозитории ветку some_branch (если её ещё нет), устанавливает связь между ней и локальной веткой some_branch и отправляет коммиты локальной ветки в облачную.

3.3. Копирует в основную ветку облачного репозитория, зарегистрированного как origin, все коммиты из локальной ветки some_branch.

3.4. Отправляет в облачный репозиторий все неотправленные туда коммиты.

§ 1.11. Запросы на слияние

В предыдущем параграфе мы добавили в наш проект новую фичу. Для этого мы создали новую ветку, создали новый файл, закоммиттили этот файл в созданную для новой фичи ветку и отправили коммит в удалённый репозиторий. Но что делать дальше? Эта фича появилась в новой ветке, но в основной ветке её нет.

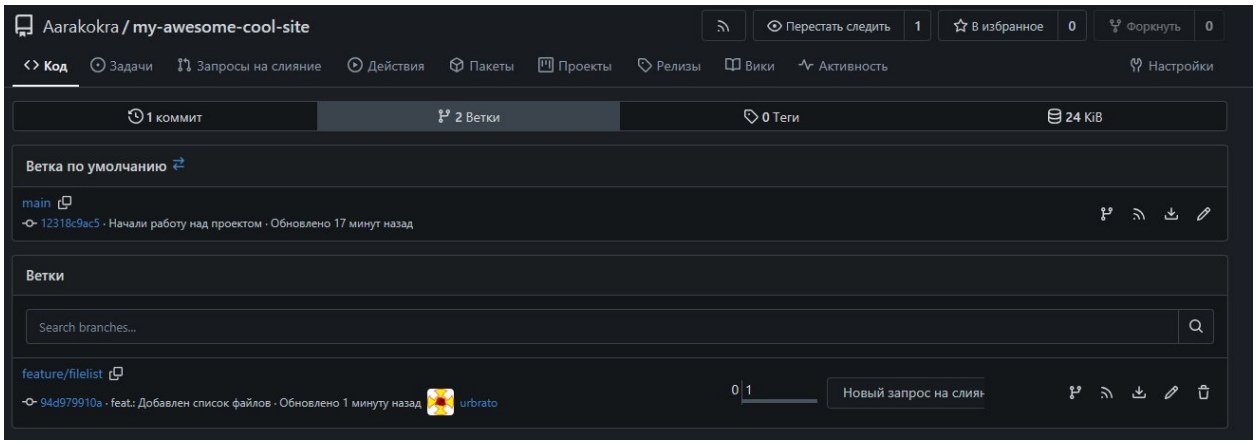
Когда над проектом работает один-два человека, из которых оба хорошие мастера, зачастую каждый из них сам сливает свои фичи в основную ветку и, если он всё же не один, предупреждает коллегу или коллег, что в основной ветке появились изменения. Однако при работе над крупным проектом обычно право на запись в основную ветку есть только у руководителя проекта, который проверяет каждую выполненную задачу, будь то реализация новой фичи или исправление ошибки, и только если новый код его устраивает, вливает его в основную ветку. Значит, нам надо как-то сообщить руководителю проекта, что мы закончили работу над новой веткой, а заодно как-то дать знать сервису удалённого репозитория, что руководитель проекта может проводить ревизию кода с возможным последующим слиянием его с основной веткой. Мы можем, конечно, написать руководителю проекта в мессенжер или позвонить по телефону, но это решает только половину задачи. Если вы решили внести свой вклад в опенсорсный проект, это тем более актуально, поскольку вы, скорее всего, даже не знаете, куда написать.

На самом деле, в сервисах удалённых репозиториев, будь то GitHub, GitLab, МосХаб или другие, есть механизм, который решает обе эти задачи: запрос на слияние. В английской версии он называется pull request в GitHub или merge request в GitLab. Оба эти названия корректны – фактически, это запрос к хозяину репозитория на то, чтобы он выполнил пулл вашего коммита (поэтому название pull request корректно) и влил его в основную ветку (поэтому название merge request также корректно). Руководитель проекта (в компании) или ответственный за поддержку проекта (для опенсорсного проекта) именно благодаря сделанному вами запросу на слияние узнает, что вы предлагаете к рассмотрению те или иные изменения. Получив запрос на слияние, руководитель проекта (будем для конкретики рассматривать такой вариант) сможет оставить к вашей работе комментарии, задать вопросы, предложить те или иные доработки и, в конце концов, в лучшем случае, одобрить сделанные вами изменения и влить их в основную ветку.

Давайте зайдём на выбранный нами сервис и посмотрим, как выглядит наш репозиторий теперь, после того, как вы отправили туда коммит в новой ветке.

Зайдите в репозиторий нашего проекта. Выберите пункт «Ветки» (его местоположение и точный вид могут различаться в разных

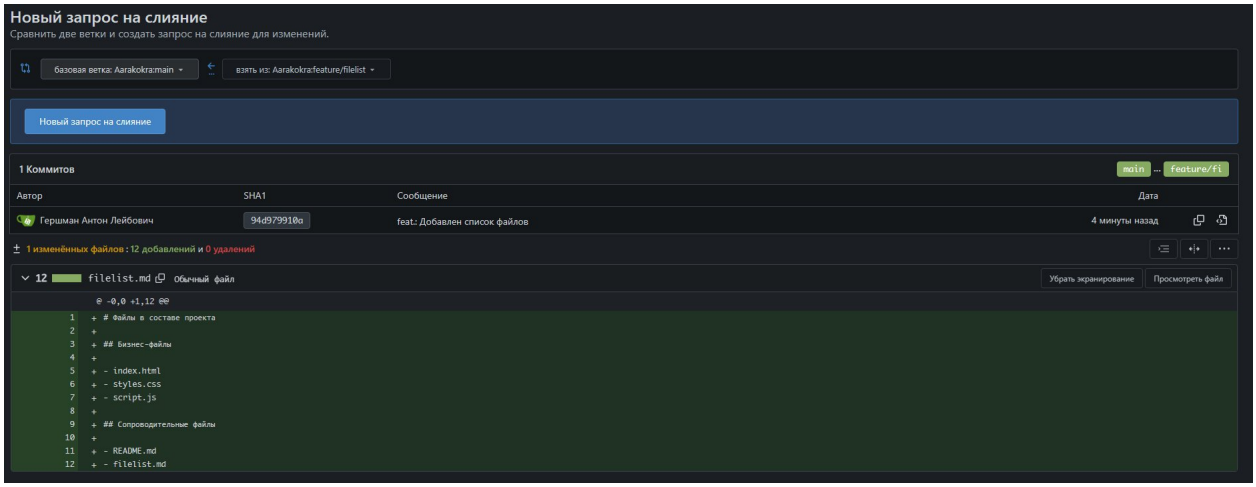
сервисах, примерно как и у пункта «Коммиты»:



В основной части окна вы увидите список веток в удалённом репозитории (то есть, если локально какая-то ветка создана, но из неё ни одного коммита не отправлено в удалённый репозиторий, то облачный сервис, конечно же, просто не будет знать о существовании этой ветки, и её в этом списке не будет).

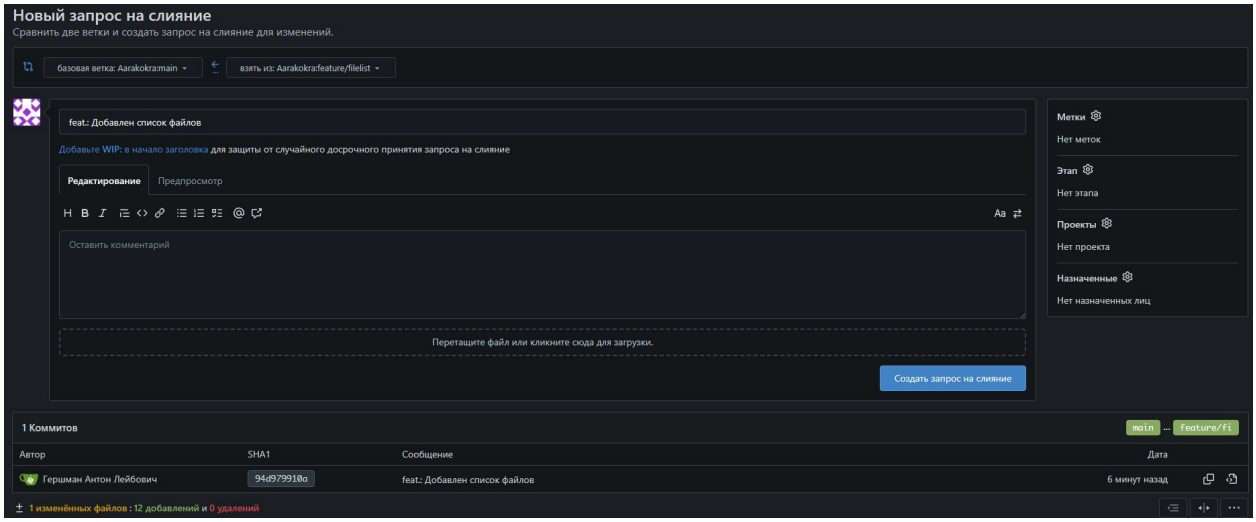
На этой иллюстрации мы видим две ветки: основную ветку main (у вас может быть master) и новую ветку feature/filelist. Ветка master помечена плашкой «по умолчанию», поскольку это основная ветка, а также плашкой «защищена», что и означает, что в неё не может пушить коммиты никто, кроме имеющего на это особые права (поскольку этот репозиторий создали вы сами, у вас эти права, конечно, есть, но другой сотрудник этих прав по умолчанию не имеет). А вот у ветки feature/filelist вы видите кнопку: «Новый запрос на слияние» (опять же, она может называться иначе).

Нажмите на эту кнопку. Появится окно создания запроса на слияние:



Вы увидите, из какой ветки в какую вливается коммит (иногда с этим путаются, так что это полезная информация). Также (в данном случае – сервис на основе Gitea – по дополнительному нажатию на кнопку «Новый запрос на слияние») здесь задаётся название запроса на слияние (по умолчанию оно совпадает с сообщением последнего коммита сливаемой ветки) и, по желанию, вводится подробное описание того, что в этой ветке делалось, зачем и почему.

Далее указывается «административная» информация – кто назначается проверяющим.



Проверяющий - это тот, кто получает запрос и кто будет, собственно, проверять ваш код, делать замечания и так далее. Если иного не предусмотрено рабочей инструкцией вашей команды, эти поля можно оставить пустыми.

В некоторых рабочих командах есть регламент, требующий одобрение запроса от одного или нескольких проверяющих прежде, нежели запрос будет принят. Это здесь также настраивается.

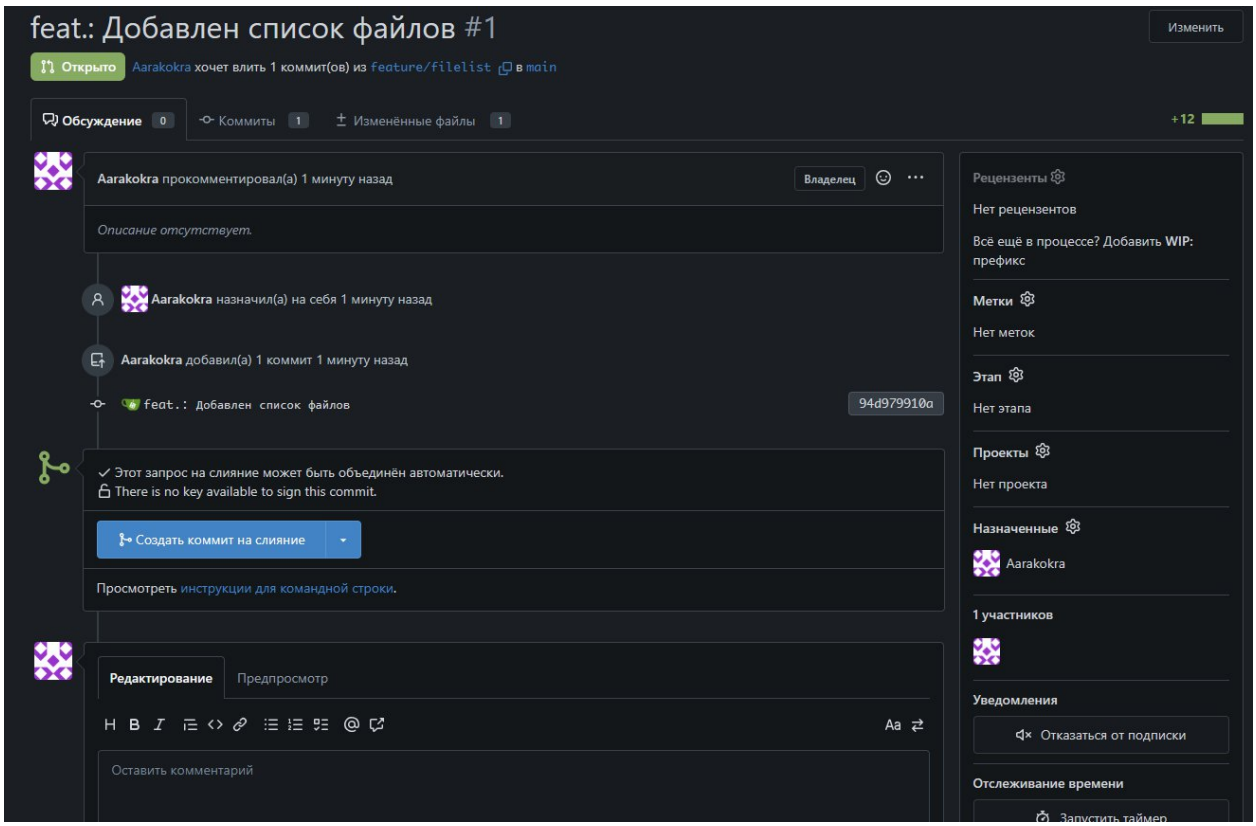
Если команда использует тот же сервис для контроля поэтапной разработки продукта, где для каждого этапа есть свои сроки, то можно выбрать этап, к которому относится данная работа. Эта опция присутствует далеко не в каждом сервисе удалённых репозиториев.

Можно запросу на слияние назначить метку, которая в списке запросов будет бросаться в глаза проверяющему: «баг», «внимание» и пр.

Наконец, в некоторых сервисах можно указать, что произойдёт после того, как слияние будет произведено. После того, как запрос на слияние принят, коммиты новой ветки окажутся в ветке по умолчанию, поэтому обычно исходную ветку, созданную под конкретную задачу, затем удаляют. Кроме того, иногда при слиянии коммитов новой ветки в ветку по умолчанию эти коммиты объединяют как бы в один большой коммит. В других сервисах может быть так, что эти опции становятся доступны уже только при принятии запроса на слияние (как, скажем, в Gitea они доступны в качестве выпадающего списка при кнопке «Создать коммит на слияние»).

Выполнив все нужные настройки, мы нажимаем кнопку «Создать запрос на слияние».

Появляется окно уже созданного запроса на слияние. Поскольку проверяющим в данном репозитории являемся мы же, то мы увидим также и те кнопки, которые доступны проверяющим:



Мы видим название запроса на слияние и названия веток, откуда и куда производится слияние. Далее идёт основная часть информации о запросе, содержащая вкладки: «Обзор», «Коммиты» и «Изменения» (для случая, когда подключены конвейеры автоматической интеграции и поставки кода, возможно наличие также вкладки «Конвейеры»).

Основная часть – это вкладка обсуждения. Здесь мы можем одобрить запрос, если рабочий регламент требует одобрений. Далее отображается ход проверки возможности слияния – может быть так, что после того коммита, от которого «отпочковалась» ваша ветка, в основную ветку были внесены коммиты, которые изменяют те же строки кода, которые изменили вы, и тогда просто так слить изменения не получится, придётся вручную разрешать конфликт. В нашем случае мы видим галочку, что запрос на слияние может быть объединён автоматически.

Также во вкладке обсуждения присутствует возможность прокомментировать запрос на слияние или задать вопрос исполнителю.

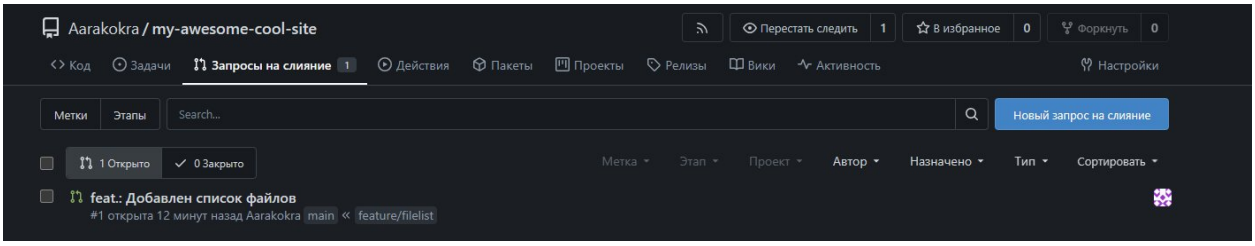
Если проверяющего не устраивают сделанные изменения, он может в комментарии указать свои пожелания по доработке. Исполнитель свои доработки оформляет коммитами в ту же сливаемую ветку. После этого облачный репозиторий автоматически перепроверяет готовность запроса к слиянию.

Вкладка «Коммиты» содержит перечень сливаемых коммитов (в нашем случае там всего один коммит), а вкладка «Изменения» позволяет проверяющему сравнить состояние сливаемой ветки с состоянием основной ветки (подобную информацию мы видели в окне создания запроса на слияние).

Наконец, вкладка «Конвейеры», когда она есть, позволяет посмотреть состояние конвейеров, отрабатывающих сделанные коммиты – мы их создание здесь рассматривать не будем (обычно созданием конвейеров занимаются люди отдельной специальности – инженеры по внедрению, или девóпсы).

В принципе, мы окно с созданным запросом на слияние увидели потому что только что этот запрос создали. А где этот запрос видит проверяющий?

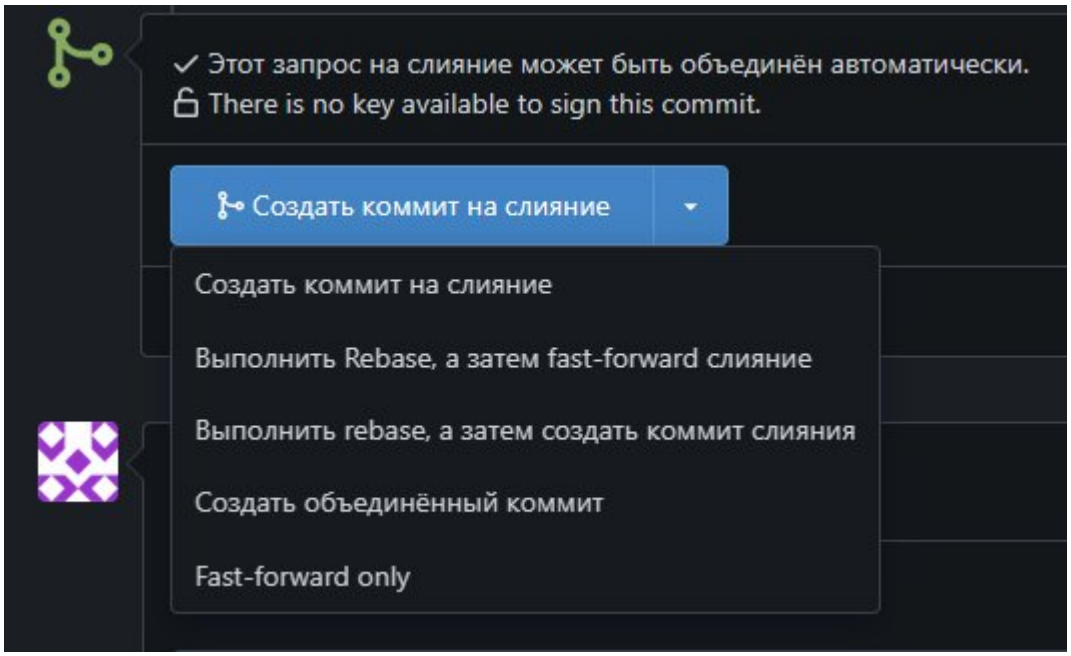
Проверяющий видит количество запросов на слияние всё в той же странице репозитория:



Выбрав этот пункт меню, проверяющий увидит список сделанных запросов на слияние, которые ещё не слиты и не отменены. Выбрав нужный запрос на слияние, проверяющий попадёт ровно в то же окно, которое мы видели.

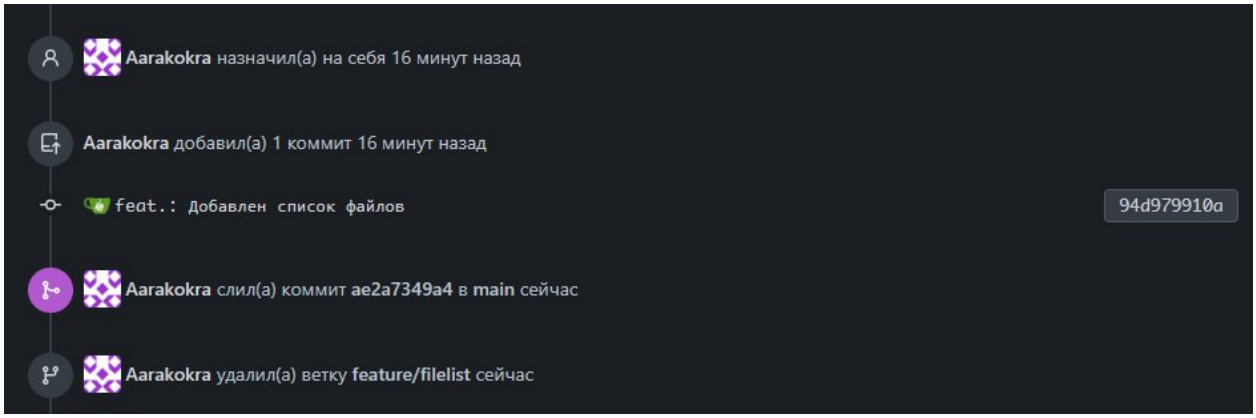
Итак, во вкладке обзора можно подтвердить запрос, нажав кнопку «Слияние», предложить доработку, оставив комментарий в поле «Активность», либо закрыть запрос, не выполнив слияние (и тем самым отвергнуть его), нажав кнопку «Закрыть запрос на слияние».

Подтвердим запрос, нажав кнопку «Слияние» (в Gitea она называется «Создать коммит на слияние» и снабжена выпадающим списком – в нём выберем также пункт «Создать запрос на слияние»).



Сервис ещё раз перепроверит возможность слияния. Если ничего не изменилось, некоторые сервисы (в том числе на базе Gitea) дают возможность отредактировать сообщение будущего коммита и решить, удалять ли влитую ветку:

После создания коммита на слияние появится подтверждение того, что слияние произведено:



Мы видим, что на удалённом репозитории сделанный коммит теперь вошёл в состав ветки main (либо master), а исходная ветка feature/filelist удалена. Как теперь эти изменения получить в локальном репозитории?

Вернёмся к Git Bash. Убедитесь, что вы находитесь в папке локального репозитория.

На удалённом репозитории ветка feature/filelist уже удалена, поэтому на локальном репозитории переключитесь на основную ветку при помощи команды git checkout или git switch. После этого просто получите последние изменения при помощи git pull, как обычно. Перед выполнением git pull обязательно убедитесь, что в вашей локальной ветке master или main нет несохранённых изменений. Выполните git status: вывод должен говорить «working tree clean». Если там есть изменения, либо закоммитьте их, либо отмените, иначе git pull может повести себя непредсказуемо.

Посмотрите на журнал изменений при помощи команды git log. После коммита, который был последним в master до слияния (точнее, фактически перед ним, поскольку более поздние коммиты выводятся раньше, чем более ранние), вы увидите коммит, который был в ветке feature/filelist, а самым недавним окажется автоматически созданный коммит из облака. Также мы видим, что между хэшкодом

коммита и информацией об авторе будет также строчка «Merge» с указанием кратких хэшей слитых друг с другом коммитов.

На самом деле, когда выполняется слияние веток через запрос на слияние, в Git есть три разные стратегии.

По умолчанию используется стратегия «Merge», или «Слияние». При этом в основной ветке создаётся коммит слияния, «merge commit», у которого не один, как обычно, а два родителя. Таким образом, из головного (последнего) коммита основной ветки появляется путь и в те коммиты, которые были в ней до слияния, и в те, которые до слияния принадлежали другой ветке. Состояние файлов в коммите слияния соответствует такому, как если бы все коммиты принадлежали одной ветке. Эта стратегия при слиянии не меняет существовавшие до слияния коммиты ни той ветки, которую сливают, ни той ветки, в которую сливают.

Другая стратегия – это «Squash and Merge», или «Объединение и слияние» («Создать объединённый коммит»). При этом все коммиты, которые должны быть влиты в основную ветку, объединяются в один большой коммит, состояние файлов в котором соответствует состоянию на момент последнего коммита сливаемой ветки, а затем этот новый коммит добавляется к основной ветке. Сливающего коммита при этом не создаётся. Эта стратегия часто используется в крупных проектах, чтобы сохранить историю основной ветки чистой: вместо десятка мелких коммитов вида «поправил опечатку», «забыл точку с запятой» в историю попадает один аккуратный коммит, описывающий всю задачу целиком.

Наконец, третий вариант – это «Rebase and Merge», «Перебазирование и слияние». В этом случае коммиты, которые должны быть влиты в основную ветку, начиная с самого старого и постепенно к самому новому, добавляются к основной ветке, как если бы соответствующие им изменения производились в основной ветке на момент после последнего коммита. Сливающего коммита при этом также не создаётся. Эта стратегия делает историю коммитов идеально линейной (как будто вы вообще не создавали отдельную ветку, а писали код прямо в master). Иногда есть подвид этой стратегии - «Rebase and fast-forward». В таком случае сливающего коммита не будет именно в этом варианте, а в варианте без fast-forward после перебазирования коммитов всё равно будет создан новый коммит.

Стратегии, отличные от «Merge», требуют осторожности: они переписывают историю коммитов, поэтому их никогда не применяют к веткам, над которыми уже работают другие люди.

Увы, не все облачные репозитории поддерживают все эти стратегии, но знать о них надо.

Давайте посмотрим на список веток в локальном репозитории. Мы видим, что, хотя в облачном репозитории ветка feature/filelist была удалена, в локальном репозитории она осталась. Чтобы её удалить, используйте ту же команду branch, но перед именем ветки используйте ключ -d (от слова «delete» – «удалить»):

```
git branch –d feature/filelist
```

Только перед этим убедитесь, что находитесь в ветке master.

Обратите внимание, что если ветка не была слита (или если она была слита в облачном репозитории, но локальный репозиторий этой информации ещё не получил), Git откажется удалять ветку, чтобы не дать вам потерять изменения. Если вдруг нужно удалить ветку с неслитыми изменениями (например, поняли, что идёте по кардинально неверному пути), вместо -d используйте -D.

На самом деле, Git умеет при получении информации из облака автоматически удалять локальные ветки, которые были удалены в облаке, но эта возможность на практике редко используется – бывает так, что локальная ветка для чего-то ещё нужна, и на такой случай эту опцию держат отключённой.

Вопросы для самопроверки:

1. Что такое запрос на слияние?

   Варианты ответов:

   1.1. Способ принудительно выполнить пуш в основную ветку.

   1.2. Запрос на внесение изменений, выполненных в отдельной ветке, в основную ветку.

   1.3. Способ редактировать файлы в удалённом репозитории напрямую.

   1.4. Инструмент для скачивания удалённого репозитория на локальную машину.

2. Что означает «слияние a в b» в запросе на слияние?

   Варианты ответов:

   2.1. Изменения, сделанные в ветке b, будут добавлены в ветку a.

   2.2. Изменения, сделанные в ветке a, будут добавлены в ветку b.

   2.3. Ветка a представляет исходный репозиторий, ветка b форк.

   2.4. Ветка a представляет форк исходного репозитория b.

3. Какой стратегии отработки запроса на слияние не существует?

   Варианты ответов:

   3.1. Merge

3.2. Squash and Merge

3.3. Rebase and Merge

3.4. Fork and Merge

Часть 2. Работа в команде

§ 2.1. Проверка кода

В первой части вы научились создавать ветку для внесения изменения, будь то реализация новой фичи или исправление ошибки, создавать запрос на слияние этой ветки с общей кодовой базой, и даже осуществлять это самое слияние. Но на практике, как правило, ответственный за поддержку проекта не проводит слияние сразу, как увидит запрос на него – мало ли что там написано: код может не соответствовать общим для рабочей группы соглашениям, код может быть неэффективным, наконец, код может просто сломать работу приложения. Поэтому, прежде чем разрешать слияние, проверяющий производит то, что в англоязычной литературе называют code review, а у нас код-ревью, ревизией кода или проверкой кода.

Как проходит проверка кода? В разных рабочих группах по-разному. Но в любом случае в этом процессе, как правило, можно выделить три основных этапа.

Прежде всего, проверяющий убеждается, что конвейер непрерывной интеграции, или CI, отработал без ошибок. Настройка конвейеров непрерывной интеграции (CI) и непрерывной поставки (CD) – это отдельная большая тема, и занимаются этим, как правило, девóпсы, или инженеры по внедрению. Однако даже если вы не умеете настраивать конвейер непрерывной интеграции, вы всё равно можете без особого труда зайти на страницу, на которой отображается процесс его работы, и убедиться, что он завершён успешно, когда это произойдёт.

Во время работы конвейера непрерывной интеграции, как правило, автоматически выполняются в некоей выделенной среде все те действия, которые выполнил бы другой сотрудник, получив ваши изменения: из Git забирается свежая версия кода, запускается компилятор (если проект подразумевает использование компилируемых языков программирования), запускаются средства автоматического тестирования, запускаются средства проверки качества кода, и т.д., и т.п. Таким образом, если девопс грамотно настроил конвейер CI, успешная отработка этого конвейера, по крайней мере, означает, что код не роняет всё приложение, не приводит к сбоям ранее успешно проходимых тестов и, если уж конвейер совсем хороший, удовлетворяет требованиям стиля кода, принятым в вашей рабочей группе.

Если конвейер завершается с замечаниями или, паче, останавливается с ошибкой, проверяющий потребует от приславшего запрос на слияние решить проблемы. Проверяющий может просто указать на имеющуюся ошибку, а может и предложить конкретное решение или задать наводящие вопросы, после которых исполнителю будет легче обнаружить и исправить недочёты. Всё зависит от стиля работы команды и сложности ситуации.

Если конвейер отработал успешно, наступает второй этап – проверяющий просматривает сделанные изменения и анализирует их. На этом этапе проверяющий может задать вопросы, почему выбран тот или иной конкретный способ решения поставленной задачи, так что исполнитель должен быть готов объяснить свои действия. Кроме того, проверяющий также может потребовать доработать предложенные к слиянию изменения – например, если какой-то алгоритм можно заменить более быстрым или менее требовательным к ресурсам.

Наконец, на третьем этапе проверяющий, как правило, вытягивает предложенную к слиянию ветку локально и запускает приложение с предложенными изменениями – проверяет, как оно будет с ними работать, не «поломалась» ли какая-то функциональность, всё ли работает ожидаемым образом и т.д.

Как проверяющий вытягивает ветку локально? Если он просто сделает git pull *имяветки*, то, скорее всего, ничего хорошего не получится. Для начала проверяющий должен обновить в локальном репозитории сведения о ветках, залитых в облако, командой git fetch origin. Эта команда скачает на локальную машину текущее состояние облачного репозитория во всех ветках, но в тех ветках, которые локально уже существовали, не сольёт эту информацию с локальными данными, а сохранит отдельно. После этого, когда локальный репозиторий уже содержит нужную ветку, можно будет переключиться на неё командой git switch *имяветки*. Если же вы работали в своём форке, то проверяющему сначала нужно будет добавить его как дополнительный удалённый источник и получать ветку уже оттуда.

Если проверяющего всё устраивает, он может, если это используется (обычно, если проверяющих больше одного) нажать кнопку «Одобрить» – возможно, с традиционным комментарием «LGTM», что расшифровывается как «looks good to me», т.е. «как по мне, выглядит нормально». После того, как необходимое количество проверяющих одобрило запрос (или сразу, если проверяющий один), можно производить слияние.

Сколько проверяющих должно одобрить запрос? Зависит от регламента, принятого в рабочей группе. Где-то достаточно одного, где-то должны одобрить все, где-то любые двое... Это уже детали.

НА ЗАМЕТКУ:

Помимо LGTM, в комментариях к запросам на слияние часто встречаются и другие сокращения:

- ACK (acknowledge) – «принято, согласен»
- NACK (negative acknowledge) – «не согласен, отклоняю»
- WIP (work in progress) – «работа в процессе, пока не готово к слиянию»
- RFC (request for comments) – «запрос комментариев, нужно обсуждение»

Знание этих аббревиатур поможет вам быстрее ориентироваться в переписке по запросам на слияние.

Что происходит, если запрос на слияние не одобрен? В таком случае проверяющие высказывают своё мнение в комментариях к запросу. Иногда это просто комментарий, в котором проверяющие высказывают пожелания, что именно следует изменить, но не ограничивают исполнителя в способах, а иногда это комментарий с конкретным запросом на изменения. В любом случае, пока эти изменения не будут внесены, проверяющий не одобрит запрос.

После того, как исполнитель получил таким образом обратную связь, он вносит доработки в ту же ветку, запрос на слияние которой он отправлял. После каждого коммита запрос на слияние автоматически обновляется, и проверку кода требуется выполнить заново.

В некоторых сервисах удалённых репозиториев проверяющий может сам предложить конкретные изменения в сливаемую ветку, а исполнитель прямо в веб-интерфейсе эти изменения может закоммитить, так, как если бы он их сделал сам. Однако если предлагаются достаточно серьёзные изменения, возможно, имеет всё же смысл внести их локально, как следует проверить, а затем уже скоммитить.

Всё, сказанное в этом параграфе, конечно же, относится и к опенсорсным проектам. Но в этом случае следует помнить, что любой опенсорсный проект поддерживают обычные люди, которые тоже могут ошибаться, которые тоже могут уставать, и у которых это, как правило, не основная деятельность, за которую они получали бы зарплату, а значит, они ей занимаются в свободное время. Таким образом, не нужно расстраиваться, если проверка вашего вклада в опенсорсный проект занимает больше времени, чем вы надеялись. Не стоит терроризировать владельцев репозитория частыми напоминаниями о вашем запросе. Наберитесь терпения, и оно будет вознаграждено. Если ваш запрос не рассматривают больше недели-двух, можно вежливо написать один комментарий. Например (для международного проекта с открытым исходным кодом): «Hi! Just checking if there's any feedback on this PR. Happy to make any necessary changes. Thanks!» Это покажет вашу заинтересованность без навязчивости.

Также помните об уважении к коллегам. Старайтесь не выкладывать недостаточно проверенный код и уделяйте должное внимание принятым на проекте соглашениям о стиле кода. Перед отправкой запроса на слияние задайте себе несколько вопросов:

- Запустил ли я тесты локально?
- Проверил ли я, что конвейер CI/CD прошёл успешно?
- Соответствует ли мой код соглашениям о стиле?

- Понятны ли имена добавленных мной имён типов, функций и переменных?
- Добавил ли я комментарии к сложным местам?

Если вы ответили утвердительно на все вопросы, код готов к проверке.

Вопросы для самопроверки:

1. Каков обычно первый шаг проверки кода?

   Варианты ответов:

   1.1. Просмотр предлагаемых изменений.

   1.2. Проверка результата работы конвейера непрерывной интеграции.

   1.3. Получение предлагаемых изменений локально.

   1.4. Одобрение запроса на слияние.

2. Что означает комментарий «LGTM»?

   Варианты ответов:

   2.1. Let’s Get This Merged (давайте сливать это)

   2.2. Looks Good to Me (как по мне, выглядит нормально)

   2.3. Let’s Go to Manual (давайте посмотрим справочную информацию)

   2.4. Long-term Goal to Maintain (в долгосрочную поддержку)

§ 2.2. Когда проверяющий – это вы

В прошлом параграфе мы провели общий обзор того, как производится проверка кода. Когда вы исполнитель, этого, в принципе, достаточно для того, чтобы понимать, что происходит между созданием запроса на слияние и его закрытием (со слиянием или без него). Но что если проверяющий – это вы? На что, собственно, ориентироваться при проверке кода?

Прежде всего, нужно понимать, что проверка кода – это не сеанс показательной порки. Мы проводим проверку кода не для того, чтобы самоутвердиться за счёт исполнителя, а для того, чтобы совместно создать максимально качественный код, и для того, чтобы поучиться друг у друга, войти в курс работы друг друга и, таким образом, уменьшить фактор автобуса[1]. Если превращать проверку кода в свару между исполнителем и проверяющим, это приведёт только к ухудшению морального климата в коллективе, ухудшению качества последующих проверок и, в конечном счёте, больно ударит по качеству продукта.

Проверяющий (разумеется после завершения первого этапа – подтверждения, что интеграция кода проходит успешно) должен, прежде всего, понять задачу, которая была поставлена перед исполнителем, определить, ту ли задачу он решил, и, собственно, вчитаться в код, который исполнитель предложил в результате. Работа проверяющего тут не менее ответственная, чем исполнителя – он должен оценить метод, которым исполнитель решал поставленную задачу, удостовериться в его адекватности этой задаче и проверить качество его реализации.

Затем проверяющий по возможности должен дать обратную связь. Чисто психологическая рекомендация, выверенная на практике – лучше, если проверяющий работает не в парадигме «вот это и это было плохо, а вот это и это нужно поправить», а в парадигме «вот это было сделано хорошо, а вот это можно сделать лучше, а именно вот так» – да-да, это классическая модель развивающей обратной связи (известная в менеджменте как «Плюс/Дельта» или «Start/Stop/Continue»). Она с самого начала настраивает исполнителя на позитивный лад, таким образом поощряя его воспринимать комментарий проверяющего не как строгие замечания, на которые хочется возразить, даже если нет аргументов, а как предложение возможностей к самосовершенствованию (собственно, поэтому данная модель на практике и оказалась лучшей для организации комментариев проверяющего).

Далее – если реакция проверяющего зависит от его настроения, от отношения к исполнителю, от фазы Луны или уровня кофеина в крови, это плохо и, не побоюсь этого слова, шизогенно. Исполнитель, взаимодействующий с таким проверяющим, не может сделать адекватный вывод о том, как следует совершенствовать свой

[1] Фактор автобуса (bus factor) – условная оценка ущерба, который будет нанесён проекту в случае, если кого-то из участников собьёт автобус.

код, и вместо улучшения качества мы на выходе получаем перебор вариантов реализации вслепую. Процесс проверки кода должен быть понятен всем участникам проекта, критерии должны быть чётко сформулированы и понятны каждому.

В идеальном случае во внутренней документации команды есть чек-лист – чёткий и однозначный список требований ко всем проверяемым параметрам кода: к оформлению, к структуре, к выбору алгоритмов и т.д. Проверяющий может ссылаться на пункты чек-листа и, таким образом, аргументировать свои предложения по улучшению кода. Конечно, идеальных случаев на практике практически не бывает, и какой-нибудь очередной запрос на слияние может вскрыть противоречие между двумя разными пунктами чек-листа. В этом случае исполнитель также может сослаться на конкретные пункты и, совместно с проверяющим, выработать решение.

Следует помнить, что если запрос на слияние содержит очень много изменений, это профанирует всю идею проверки кода – никто не сможет быстро и качественно проверить изменение, которое нельзя целиком уместить в голове. Поэтому задачу по возможности следует дробить на небольшие подзадачи, решение каждой из которых можно окинуть внутренним взором. Если руководитель при постановке задачи не озаботился её декомпозицией, дробление изменений на атомарные коммиты становится задачей исполнителя, а он не всегда может быть достаточно компетентен в этом вопросе. Поэтому данная рекомендация относится в большей степени к лиду и к проверяющему, но в какой-то степени касается и исполнителя (а в случае работы над проектом с открытым исходным кодом она касается исполнителя в первую очередь). В разных командах могут встречаться разные рекомендации на этот счёт – например, воздерживаться от подачи запроса на слияние, изменяющего больше десяти файлов и имеющего в своём составе суммарно более двухсот добавленных, изменённых или удалённых строк. Если изменение оказывается слишком большим, исполнитель может разбить её на подзадачи, каждая из которых требует не очень большого запроса на слияние. Например, добавление новой фичи иногда можно разбить на стадии доработки структуры базы данных, поддержки добавленных данных в слое взаимодействия с базой данных, реализации бизнес-логики, добавления новых методов в API бэкенда и, в последнюю очередь, внесения изменений во фронтенд (что тоже можно разбить на стадии дизайна новых окошек, добавления их логики, добавления вызовов бэкенда и, наконец, добавления вызова этих окошек).

Проверяющий должен по возможности избегать «замыливания взгляда». Поэтому не стоит проверять запрос на слияние слишком долго. Если на проверку кода требуется более получаса-часа (проверяющие тоже разные бывают), следует отвлечься на другие задачи и вернуться к запросу через какое-то время. Чтобы сэкономить время на погружении проверяющего в задачу, исполнитель в комментарии к запросу на слияние (или к задаче в трекере задач) может описать, что именно было сделано, каким образом и по каким соображениям. Если исполнитель сам сомневается в каком-то своём решении, также будет хорошей идеей отразить это в своём комментарии, так проверяющий быстрее обратит

внимание на сомнительный момент и, может быть, даст какую-то конкретную рекомендацию по его поводу в контексте данной задачи или на будущее.

Не имеет смысла править оформление кода, если что-то не в порядке с методикой решения задачи – в процессе правки реализации код всё равно может быть серьёзно переписан, и оформление придётся проверять фактически заново. Поэтому проверку кода следует проводить от более глубоких пунктов чек-листа (если он есть) к более мелким: сначала проанализировать подход исполнителя в целом, затем погрузиться в детали реализации, а стиль кода оставить «на десерт».

Идеально, если проверяющий разбирается в характере исполнителя. Кому-то комфортнее, когда замечания подаются поочерёдно – сначала более важные, потом (с обязательным упоминанием, что предыдущие замечания отработаны хорошо) более мелкие. Такого исполнителя не стоит приводить к совершенному коду за один присест, он так решит, что никуда не годен как разработчик, и быстро сломается. Кому-то, наоборот, комфортнее получить сразу весь (или хотя бы предположительно весь) список замечаний, которые можно отработать, а поочерёдная подача вызовет у него ощущение, будто бы к нему придираются, каждый раз находя что-то новое. Поэтому лучше всего, если исполнитель и проверяющий уже имеют опыт совместной работы (это следует учитывать менеджеру, когда он формирует команду под конкретный проект). Особенно актуальны такие нюансы, когда между исполнителем и проверяющим имеется большой разрыв в уровне. Проверяющему следует помнить, что если ему пришёл на проверку код, написанный, условно говоря, на двойку, то вряд ли получится заставить исполнителя сразу сделать из него код, написанный на пятёрку. Если удастся вытянуть на тройку – уже хорошо. Через две-три задачи, может быть, тройка станет уже базовым уровнем, и можно будет натягивать на четвёрку. А там, глядишь, уже можно будет и твёрдого хорошиста превращать в отличника. Работа в команде – это всегда в какой-то степени взаимодействие мастера и подмастерья. Этот принцип наставничества – историческая Белл-Ланкастерская система взаимного обучения, освящённая веками функционирования гильдий вольных каменщиков – доказанно приводит к появлению большего количества мастеров, что всегда положительно влияет на отрасль в целом.

Не следует в обратной связи давать множество однотипных комментариев, типа: «вот тут используй using вместо try/finally... И тут... И вот тут... И здесь ещё». В разработке есть отличный принцип – DRY (don’t repeat yourself, не повторяйся). Следует иметь его в виду и при проверке кода. Вполне достаточно указать на один конкретный случай и попросить самостоятельно выявить и исправить аналогичные моменты.

Наконец, проверяющему следует помнить – этап проверки кода блокирует задачу, в рамках которой был выполнен запрос на слияние. Условно говоря, «мячик у него в руках», и теперь время внедрения изменений зависит от него. Поэтому, как бы этап проверки кода ни казался чем-то мелким и неважным, затягивать с проверкой сверх необходимого нельзя. Это не означает, что качеством проверки

можно пренебречь – это означает, что не следует откладывать её в долгий ящик.

Вопросы для самопроверки:

1. Какие задачи **не** решает проверка кода?

   Варианты ответов:

   1.1. Уменьшение зависимости проекта от выхода из строя конкретного исполнителя.

   1.2. Улучшение качества кода.

   1.3. Установка иерархии и дисциплины в команде.

   1.4. Повышение уровня исполнителей.

2. В каком порядке следует давать обратную связь?

   Варианты ответов:

   2.1. Сначала общие замечания, затем конкретные.

   2.2. Сначала анализ общего уровня исполнителя, затем упоминание влияния недостатков исполнителя на конкретные места предлагаемого к слиянию кода.

   2.3. Сначала перечисление, что было сделано хорошо, а затем предложение, что можно сделать лучше.

   2.4. Сначала указание на срок сдачи проекта, затем анализ скорости работы исполнителя.

3. Расставьте предлагаемые изменения в правильном порядке:

   Изменения:

   3.1. Невнимание к стилю именования переменных.

   3.2. Ошибки в структуре кода.

   3.3. Выбор неверного алгоритма в реализации одного из новых методов бизнес-логики.

   3.4. Нехватка самодокументирующих комментариев.

§ 2.3. Синхронизация репозиториев

Когда несколько человек работают над одним проектом, вопрос синхронизации результатов работы встаёт особенно остро. Мы рассмотрим сценарий, когда у вас есть локальный репозиторий, а у команды в целом есть удалённый репозиторий, а также рассмотрим отдельно ситуацию, когда ваш проект является форком более глобального проекта и, таким образом, кроме локального репозитория у каждого участника и общего удалённого репозитория команды, есть ещё и источник, от которого сделан форк (как мы помним, удалённый репозиторий в Git обычно обозначают как origin, а источник в случае форка проекта обозначают как upstream).

В большинстве случаев вы участвуете в оригинальном проекте, и в этом случае вам приходится синхронизировать между собой содержимое только двух репозиториев – локального и удалённого.

Когда вы клонируете удалённый репозиторий на свой компьютер, вы тем самым создаёте связку из двух элементов: удалённого репозитория в облаке (origin) и рабочей копии на своей машине. Соответственно, синхронизация между ними тоже двусторонний процесс: вам нужно получить изменения из удалённого репозитория, встроить их в свой локальный репозиторий, а изменения, сделанные локально, отправить в удалённый репозиторий.

Прежде, чем получать изменения из удалённого репозитория, нужно, чтобы состояние рабочих файлов в репозитории не отличалось от их состояния на момент последнего коммита (не считая, конечно, игнорируемых файлов). То есть, прежде всего, нужно выполнить коммит. Как это делать, вы уже знаете в подробностях.

Но что делать, если некоторые изменения, которые мы внесли, коммитить ещё рано, или мы просто не хотим их коммитить прямо сейчас? В таком случае, можно их «спрятать» – сделать так, чтобы эти изменения пропали из рабочего состояния, но сохранились в отдельном хранилище локального репозитория, своего рода тайнике. Для этого служит команда

```
git stash
```

По умолчанию git stash прячет только изменения в файлах, которые Git уже отслеживает. Если вы создали новый файл и ещё не добавили его в индекс командой git add, git stash его не тронет. Чтобы спрятать абсолютно всё, включая новые файлы, используйте команду с ключом -u (от untracked):

```
git stash -u
```

или

```
git stash --include-untracked
```

Потом, когда мы выполним всю синхронизацию, и захотим вернуть спрятанные изменения в рабочее состояние, мы сделаем это командой

```
git stash pop
```

На самом деле, есть один нюанс. Команда git stash pop возвращает спрятанные изменения в рабочую папку, удаляя их из тайника. Однако, если в процессе восстановления ранее спрятанных изменений что-то пошло не так, из тайника они всё равно исчезнут. Поэтому правильнее использовать последовательность из двух команд. Сначала восстанавливаем спрятанные изменения, не удаляя их из тайника:

```
git stash apply
```

И только если восстановление изменений завершилось успешно, удаляем их из тайника командой

```
git stash drop
```

Таким образом, если вдруг что-то пошло не так, мы не потеряем нашу работу.

Итак, допустим, у нас нет незакоммиченных, неспрятанных и неигнорированных изменений, и мы готовы к синхронизации рабочего репозитория с удалённым.

Для того, чтобы свежие коммиты из origin появились в вашем рабочем репозитории, есть несколько способов. Один из них нам уже знаком – это выполнение команды

```
git pull
```

В наиболее безопасном сценарии, когда вы ведёте работу над своей задачей в отдельной ветке, а ветка по умолчанию (master или main) служит для хранения полностью рабочего на текущий момент кода, соответственно, нужно сначала переключиться в эту ветку:

```
git switch master

git pull
```

Кроме того, возможна ситуация (при клонировании вряд ли, но в общем случае бывает), когда у локального git нет информации о том, из какой ветки и какого удалённого репозитория нужно получать коммиты в текущую ветку. В таком случае локальное имя, данное удалённому репозиторию, и имя ветки также надо указать:

```
git switch master

git pull origin master
```

Чтобы говорить о втором способе, немного обсудим, как работает команда git pull. На самом деле эта команда выполняет два действия одно за другим, и для каждого из этих двух действий есть отдельные команды.

Первое, что делает команда git pull – это получает из origin (или другого источника, если это указано в параметрах команды) свежие коммиты. Для этого служит отдельная команда git fetch. Мы можем выполнить её без параметров, тогда она использует origin в качестве источника (как и pull) и загрузит все коммиты всех веток этого репозитория. Чтобы загрузить коммиты только нужной ветки, можно также указать эту ветку (что и делает git pull):

```
git fetch origin master
```

У команды git fetch есть также ключ --all, который заставит извлечь коммиты из всех веток и из всех зарегистрированных для вашего репозитория удалённых репозиториев, и ключ --dry-run, который не скачивает сами коммиты, но выводит на экран то, что вывела бы команда без этого ключа.

Коммиты, полученные по команде git fetch, сами по себе не попадут в вашу ветку master (или любую другую вашу локальную ветку). Вместо этого, они сохранятся отдельно в ветки с префиксом в виде имени ссылки на удалённый репозиторий (например, для ветки master из репозитория origin такая ветка будет называться origin/master). По умолчанию такие ветки не отображаются в списке веток (их можно показать в списке, но для этого нужно команде git branch указать ключ -r, от слова remote).

После того, как удалённые коммиты скачались в локальную специальную ветку (в нашем случае origin/master), команда git pull выполняет слияние командой

```
git merge origin/master
```

Команда git merge выполняет слияние, вливая в текущую ветку те коммиты из ветки, указанной в качестве параметра, которых в текущей ветке нет. В данном случае у вас текущая ветка master, а ветка, чьи коммиты нужно влить – это origin/master, которая только что была получена при помощи git fetch. Разумеется, если в команде pull была указана другая ветка, то и в merge тоже попадёт то название, которое было указано.

В случае, если в вашей локальной ветке master нет таких коммитов, которых не было бы в удалённом репозитории, слияние происходит методом fast-forward («быстрой промотки») – свежие коммиты из origin/master просто добавляются один за другим в вашу ветку master, как если бы вы сделали их локально (однако, с сохранением первоначального авторства и времени создания, разумеется).

Однако, часто бывает, что за время, прошедшее с последней синхронизации, вы тоже успели сделать какие-то коммиты в свой локальный репозиторий (собственно, наиболее частый сценарий в этом и состоит: вы выполнили какой-то кусочек работы, скоммитили его локально и теперь синхронизируетесь с удалённым репозиторием). В таком случае слияние происходит так же, как происходило бы при удовлетворении запроса на слияние – создаётся коммит слияния, у которого два родителя – один из вашей локальной ветки master, другой из ветки origin/master, и который объединяет в себе изменения, сделанные в одном и в другом коммите.

Второй способ получения в локальный репозиторий свежих коммитов из удалённого репозитория, собственно, состоит в том, что мы, вместо того, чтобы выполнять команду pull, выполняем вручную сначала команду git fetch, а затем команду git merge. Это запутаннее, чем просто выполнить git pull, но даёт больше контроля над процессом – иногда это может пригодиться.

Ситуация, когда в вашем master есть коммиты, которых нет в origin/master, а в origin/master есть коммиты, которых нет у вас, называется расхождением истории (history divergency). До тех пор, пока эти коммиты не затрагивают одни и те же строки кода, в расхождении истории ничего страшного нет – вы получаете отсутствовавшие у вас коммиты, сливаете их со своей веткой master, а затем выполняете пуш обратно в удалённый репозиторий вместе со своими изменениями. Однако бывают случаи, когда изменения в master и в origin/master взаимоисключающие (например, вы в какой-то строчке добавляете к переменной единицу, а в оригинале в этой строчке переменная увеличивается вдвое, или, скажем, вы внесли изменения в какой-то файл, а в оригинале файл вообще удалён), расхождение истории вызывает конфликт между основной веткой у вас и основной веткой в оригинале, и разрешение этого конфликта может стать довольно сложным делом (разрешение конфликтов мы ещё будем рассматривать отдельно). В этом случае коммит слияния не будет создан, пока вы не разрешите все конфликты локально.

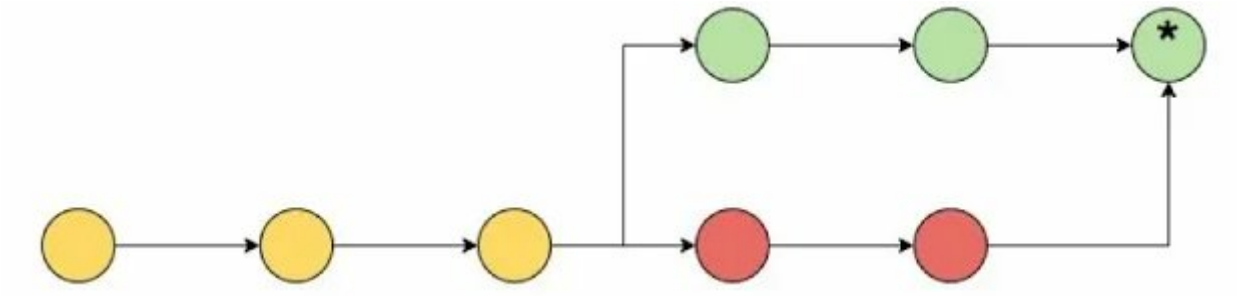
Что нужно делать, чтобы уменьшить вероятность конфликта? Наилучший вариант – не вносить коммиты в ветку по умолчанию вручную, а делать это только через подтверждение запросов на слияние. В этом случае основная ветка (master или main) служит как бы зеркалом удалённого репозитория, а все ваши доработки производятся в отдельных ветках.

Есть и ещё один вариант получения изменений из удалённого репозитория в свою локальную ветку – не слияние, а перебазирование. Для этого вместо команды git merge используется команда git rebase:

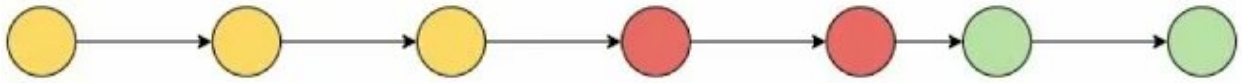
```
git rebase origin/master
```

Что при этом происходит? Git ищет самый свежий из коммитов вашей ветки master, который ещё присутствует в origin/master, и затем к нему начинает дописывать более новые коммиты из origin/master, а вот после них уже ставит те коммиты, которые до этого шли после найденного коммита. То есть, если представить, что origin/master – это ствол дерева, а git – это садовник, то git находит то место, где от основного ствола отходят ваши изменения, отрезает их и перепрививает к самой макушке.

То есть, при слиянии история коммитов выглядит примерно так (мы здесь жёлтыми кружками обозначаем коммиты, которые были и в origin/master, и в master, красными кружками свежие коммиты origin/master, зелёными свежие коммиты master):



А при перебазировании – примерно так:



Что это даёт? Более линейную историю. Коммиты в ней могут идти не в хронологическом порядке, но в целом она будет смотреться логичнее и понятнее.

Кстати, перебазирование можно сделать с любой веткой, «пересадив» её на новое место. Например, если мы дисциплинированно ведём разработку новой фичи в отдельной ветке feature/something, а в origin/master за это время появилось какое-то изменение, которое нам было бы очень неплохо учесть в разработке, мы можем выполнить

```
git switch master
git fetch origin master
git rebase origin/master
git switch feature/something
git rebase master
```

Таким образом, мы получаем свежие изменения из origin/master, перебазируем наш master на свежий origin/master, а затем перебазируем наш feature/something на свежий master.

Разумеется, при перебазировании также может возникнуть конфликт, и его тоже нужно будет разрешать. Кстати, если при запросе на слияние вашей ветки с origin/master возникнет конфликт, оптимальный способ его разрешить – это опять же выполнить слияние вашей ветки с master (обновлённый свежими изменениями из origin, разумеется), только слияние при этом не в master, а в вашей ветке или перебазирование вашей ветки на master. После того, как вы разрешите возникшие при перебазировании конфликты, они исчезнут и из запроса на слияние (если в master до этого не будет добавлено ещё какого-нибудь конфликтогенного коммита).

Команда git pull тоже может вторым этапом вместо merge делать rebase – для этого нужно указать ключ --rebase либо в конфигурации git (на требуемом вам уровне) установить значение pull.rebase в true. Например:

```
git config --global pull.rebase true
```

Эта настройка делает ваш git pull по умолчанию сохраняющим линейность истории (как fetch + rebase), что является рекомендуемой практикой во многих современных командах. Однако, если вы работаете в ветке, которую уже отправили в облако и которую используют другие, лучше временно отключить это или использовать fetch + обычный merge.

Если вам нужно делать rebase при пулле только изредка, правильнее будет воспользоваться ключом только в конкретной ситуации. Если же перебазирование – ваша основная стратегия, то правильнее будет настроить Git на постоянное использование перебазирования.

Важно иметь в виду, что команда rebase меняет те коммиты, которые перебазирует на новое место (ведь у них меняется указание на родителя), поэтому это более или менее безопасно делать только для тех коммитов, которые ещё не были добавлены в удалённый репозиторий. Перебазировать коммиты более ранние, чем предыдущий пуш, крайне не рекомендуется, это может привести к очень тяжело исправляемым последствиям. Для таких ситуаций нужно использовать merge. Исключение – если с вашей веткой больше никто не работает. В таком случае можно при выполнении пуша перезаписать всю историю ветки на удалённом репозитории её историей из локальной версии. Для этого используется ключ --force («силой») команды git push. Ещё раз – не используйте без согласования с руководителем проекта этот ключ, если с веткой, которую вы пушите с этим ключом, работает кто-то, кроме вас – это приведёт к тяжёлым последствиям. Для локальных веток, которые ещё не пушились, rebase безопасен.

А что делать, если конфликт всё же возник? Как его разрешать?

Если в процессе слияния или перебазирования возник конфликт, процесс останавливается, а Git указывает, в каких файлах имеется конфликт. Если мы откроем какой-нибудь из этих файлов, мы увидим, что в спорных местах (где в одном и том же месте сделано два разных изменения в разных ветках) он принял примерно такой вид:

```
Общая часть текста до конфликта
<<<<<<< HEAD
Изменения в текущей ветке – например, в master
=======
Изменения во входящей ветке – например, в origin/master
>>>>>>> имя ветки – например, origin/master
Общая часть текста после конфликта
```

Таким образом, у нас есть вся информация для того, чтобы понять, каким образом изменения друг другу противоречат, и как это исправить.

Каждый из файлов, про которые Git сказал, что там имеется конфликт, нужно открыть в редакторе, прошерстить на наличие таких включений и отредактировать так, чтобы файл был рабочим.

После того, как все конфликтные файлы исправлены, нужно добавить их в индекс (обычной командой git add) и продолжить слияние командой

```
git merge --continue
```

или перебазирование командой

```
git rebase --continue
```

Если сливаемых или перебазируемых коммитов больше одного, конфликты в процессе работы могут возникать несколько раз, в том числе в одном файле, к этому тоже нужно быть готовым.

Кстати, если в процессе разрешения конфликта вы запутались, случайно удалили не тот кусок кода или просто хотите начать всё сначала, не бойтесь. Вы можете в любой момент прервать процесс перебазирования и вернуть ветку в то состояние, в котором она была до начала конфликта, командой

```
git merge --abort
```

или

```
git rebase --abort
```

соответственно.

Итак, вы получили свежие коммиты из удалённого репозитория, выполнили слияние или перебазирование, разрешили все конфликты. Это половина синхронизации – теперь у вас в локальной копии ветки есть всё, что есть в этой ветке в удалённом репозитории. Осталось выполнить вторую половину – сделать так, чтобы и в этой ветке в удалённом репозитории было всё, что есть в её локальной копии.

По счастью, это намного проще – вы просто выполняете команду

```
git push
```

Если необходимо, нужно указать, куда производится пуш – в какой из удалённых репозиториев и в какую ветку в нём:

```
git push origin master
```

Единственно, что может в этой команде пойти не так (кроме случаев, когда чисто физически пропала связь с удалённым репозиторием, конечно) – если за время от пулла (или фетча) до пуша в синхронизируемую ветку в удалённом репозитории были запушены другие коммиты. Тогда перед отправкой сделанных изменений просто надо ещё раз получить изменения из удалённого репозитория в локальный.

В общем и целом, синхронизацию репозиториев мы рассмотрели. Теперь обратим внимание на ситуацию, когда репозиториев, с которыми мы работаем, не два, а три – когда наш проект является форком какого-то другого, и таким образом, у нас в локальном репозитории есть связь и с нашим удалённым репозиторием (обычно под именем origin), и с источником, от которого выполнен форк (обычно под именем upstream.

В этом случае мы крайне редко пишем в upstream (или вообще никогда – мы развиваем свой форк, который может уже слишком сильно отойти от источника, чтобы у его авторов возникло желание синхронизироваться с вами), но вполне может возникнуть ситуация, когда мы хотим в наш форк внедрить какое-то изменение, выполненное в источнике.

Давайте отработаем синхронизацию форка с оригиналом на практике. Выберите какой-нибудь публичный репозиторий, создайте приватный форк (в веб-интерфейсе сервиса удалённого репозитория), склонируйте форк себе локально (при помощи команды git clone), и поставьте оригинал как дополнительный удалённый репозиторий при помощи команды

```
git remote add upstream ssh-ссылка
```

Здесь upstream – это имя, традиционно присваиваемое ссылке на удалённый репозиторий, из которого сделан форк, а ssh-ссылка – это ссылка на этот репозиторий, такая же, как для клонирования. Будьте внимательны – здесь должна стоять ссылка на источник, а не на ваш форк!

Проверьте, что список удалённых репозиториев соответствует ожидаемому, выполнив команду

```
git remote –v
```

Создайте локально новую ветку, внесите какое-нибудь изменение и сделайте коммит. Если вы сделаете пуш этой ветки командой **git push –u origin** *ветка*, он уйдёт в ваш форк, оригинал останется нетронутым.

Как теперь получить в основную ветку изменения, которые, может быть, за это время накопились в оригинале?

Прежде всего, перейдём в нашу основную ветку уже известной командой

```
git switch master
```

(в предположении, что ваша основная ветка называется master). Дальше вам нужно получить в эту ветку изменения из оригинала. Напрашивается мысль сделать это так же, как мы делаем пулл из своего репозитория. Действительно, можно так сделать, просто после слова pull следует указать, что мы вытягиваем коммиты не из origin, а из upstream, а также, поскольку связь локальной ветки master установлена с origin/master, нужно явным образом упомянуть, что мы вытягиваем ветку master:

```
git pull upstream master
```

Однако, для большего контроля над таким новым для нас процессом, как обновление форка из источника, всё же лучше разбить процесс на два этапа:

```
git fetch upstream

git merge upstream/master
```

При этом первая команда получает из оригинала свежие изменения, но не выполняет их слияние с вашей основной веткой, а складывает их как бы в отдельную ветку – не master, а upstream/master, а вторая команда выполняет слияние уже полученных изменений.

Здесь лучше делать именно слияние, а не перебазирование – потому что ваша локальная основная ветка связана именно с вашим удалённым репозиторием, с тем, который origin, а источник, который upstream, служит лишь дополнительным источником обновлений. Поэтому rebase при работе именно с upstream может сработать непредсказуемым образом.

Источник форка – это, как правило, опенсорсный проект, а значит, весьма вероятно, что в нём свежие коммиты появляются довольно часто, и делают их люди, возможно, малознакомые. Вместе с тем, поскольку источник форка всё же не является вашим основным удалённым репозиторием, скорее всего, обращаетесь вы к нему нечасто. В сумме это приводит к тому, что конфликты возникают с большей вероятностью, а содержать могут десятки, если не сотни, конфликтующих файлов. Разрешить такое количество конфликтов вручную почти невозможно.

Если у вас нет открытого запроса на слияние вашей основной ветки, вы можете просто «откатить» изменения, которые у вас имеются в основной ветке (предварительно, конечно, убедившись, что есть другая ветка, в которой они присутствуют). Это можно сделать командой

```
git reset --hard upstream/master
```

Мы уже пользовались командой reset для того, чтобы убрать из индекса случайно добавленный туда файл. Эта команда намного шире по возможностям – она вообще служит для отката изменений в самых разных ситуациях и самыми разными способами. В данном случае мы выбираем принудительный откат с потерей изменений (флаг hard, т.е. «жёстко»), на состояние, соответствующее текущему состоянию в ветке upstream/master.

Следует учесть, что все команды т.н. принудительных действий могут оказать чудовищное и непредсказуемое для малоопытного человека действие на состояние репозитория, поэтому применять их следует в самом крайнем случае. По возможности конфликты надо всё-таки распутывать.

ВАЖНО: Перед выполнением git reset --hard убедитесь на 100%, что все ваши ценные локальные коммиты уже находятся в других ветках (например, в feature/...).

Команда --hard **безвозвратно удалит** все локальные коммиты в текущей ветке, которых нет в upstream/master. Восстановить их можно будет только через сложный механизм git reflog, рассмотрение которого находится далеко за рамками этого учебника.

После того, как вы изменили состояние основной ветки (будь это результатом обычного слияния или принудительного отката), это состояние нужно залить в ваш форк. Это делается обычным уже известным способом:

```
git push
```

Однако если вы из-за фатально разошедшейся истории были вынуждены сделать git reset, то теперь у вас практически наверняка имеется расхождение истории между локальным репозиторием и удалённым форком. Поэтому теперь в таком случае придётся принудительно привести состояние удалённого репозитория к имеющемуся у вас локальному состоянию. Это делается при помощи флага force, как мы уже упоминали ранее:

```
git push --force
```

Будьте **ПРЕДЕЛЬНО** осторожны с принудительными командами, такими, как git reset --hard и git push --force! Эти команды перезаписывают историю ваших коммитов, что может повлечь (и скорее всего повлечёт) сложности на последующих этапах работы (так, после того, как вы перезаписали историю на удалённом репозитории командой git push --force, у вашего коллеги, скорее всего, возникнут проблемы при выполнении git pull).

Однако если вы придерживаетесь правила «не портить основную ветку», при обновлении из оригинала у вас таких проблем быть не должно. Но помните о необходимости после выполнения слияния с источником делать пуш в форк. Это необходимо хотя бы для того, чтобы при смене компьютера или другой ситуации, когда вы рискуете потерять локальный репозиторий, всегда оставалась возможность получить все актуальные изменения из облака.

В заключение этого параграфа повторим основные шаги синхронизации, которые применимы практически к любой ситуации.

1. Сначала выполняем коммит имеющихся изменений. Если состояние рабочих файлов вашего локального репозитория отличается от состояния на момент последнего коммита в него, выполнить pull будет невозможно. Если по каким-то причинам коммит сделать нельзя или нежелательно, мы должны «спрятать» эти изменения командой git stash (от английского «stash» – «прятать»). После выполнения синхронизации мы сможем «достать» их обратно командой git stash pop.

2. Только после того, как состояние рабочих файлов локального репозитория не отличается от сохранённого в последнем коммите текущей ветки, получаем на рабочую машину изменения из удалённого репозитория, либо сразу сливая их с локальной версией (git pull), либо

разделяя получение изменений (git fetch) от их слияния с локальной веткой (git merge) или перебазирования на них локальной ветки (git rebase).

3. Только после того, как все коммиты текущей ветки, имеющиеся на удалённом репозитории, встроены в текущую ветку локального репозитория (слиянием или перебазированием, без разницы), отправляем на удалённый репозиторий изменения, сделанные локально (git push).

Вопросы для самопроверки:

1. Какова основная причина для того, чтобы воздерживаться от внесения изменений непосредственно в основную ветку, когда вы работаете с форком?

   Варианты ответов:

   1.1. Внесение изменений непосредственно в основную ветку приводит к дублированию коммитов.

   1.2. Возникает расхождение истории, что влечёт проблемы при получении свежих изменений из оригинала.

   1.3. Git запрещает коммиты в основную ветку, если репозиторий является форком.

   1.4. При обновлении из оригинала все ваши изменения будут автоматически стёрты.

2. Какую последовательность команд нужно выполнить, чтобы влить в основную ветку форка свежие изменения из оригинала?

   Варианты ответов:

   2.1. git switch master; git pull upstream.

   2.2. git pull upstream master

   2.3. git switch master; git fetch upstream; git merge upstream/master

   2.4. git fetch upstream/main; git merge

3. Как привести локальную основную ветку в соответствие к оригиналу, если нет запросов на слияние?

   Варианты ответов:

   3.1. git switch upstream/master

3.2. git reset --hard upstream/master

3.3. git rebase upstream/master

3.4. git clean upstream/master

§ 2.4. Соглашения

При работе в команде, как правило, обговариваются определённые правила: как именовать переменные, как распределять код по файлам, как писать документацию и комментарии, и так далее. Это более важно, чем кажется – единообразно оформленный код позволяет любому участнику команды легко читать то, что написано его коллегами, и быстро ориентироваться во всём проекте.

Свои соглашения есть и при работе с Git. Где-то жёстко соблюдается правило, что работа над каждой задачей ведётся в отдельной ветке, и только руководитель проекта может сливать изменения в основную ветку, а где-то создают отдельные ветки только для тех случаев, когда изменение может затронуть больше одного рабочего дня. Где-то широко применяется система тегов (мы эту тему ещё не затрагивали – в двух словах, тег представляет собой метку какого-то коммита, чтобы к нему всегда можно было вернуться), где-то она не используется вовсе. Где-то основная ветка служит для поставки кода заказчику, а для разработки используется ветка с названием, например, «dev» (от английского слова «development» – «разработка»), а где-то разработка ведётся в основной ветке, а для поставки кода заказчику служит ветка с названием, допустим, «prod» (аналогично, от «production»). Где-то ветку с работой по какой-то задаче сливают с основной веткой (или веткой для разработки) локально (и потом отправляют результат в облако), а где-то принято делать это только через запрос на слияние и его принятие в веб-интерфейсе.

Соглашения по организации рабочего процесса мы затрагивать не будем, так или иначе это дело каждой команды. Коснёмся здесь только соглашений по именованию веток и созданию сообщений коммитов.

Прежде всего – казалось бы, это очевидно, но, тем не менее, начинающими иногда игнорируется – при создании ветки нужно давать ей такое название, чтобы сразу было понятно, для чего она служит. Название «some-new-feature» – плохая идея, хуже будет разве что «abc123». Немного лучше – если название ветки представляет собой номер задачи в трекере задач. В случае, если кто-то удалит задачу из трекера (конечно, после того, как она успешно выполнена), информацию о предназначении ветки получить становится скачкообразно сложнее. Оптимальный способ выбора именования ветки – это формат

```
тип/задача-описание
```

где «тип» – это один из нескольких, заранее определённых в команде, кодов типа предназначения ветки, «задача» – это номер задачи в трекере (если в качестве трекера используется JIRA, номер, как правило, включает в себя также краткий код проекта, это ещё лучше – а если трекер поддерживает только чисто цифровую нумерацию задач, не возбраняется для именования ветки добавлять такой код самостоятельно – но опять же, эти коды должны согласовываться в

команде), а описание – очень-очень краткое (буквально в несколько слов) описание того, что конкретно будет выполнено в этой ветке.

Например, один из популярных наборов кодов типа предназначения ветки выглядит так:

- chore (от слова «chore» – «рутинная обязанность») – действия, не меняющие продукт напрямую, но нужные для рабочего процесса (настройка сборщика, смена версии используемой библиотеки и т.д.)
- ci (от «CI/CD», или «continuous integration / continuous deployment» – «непрерывная интеграция / непрерывная поставка») – действия по настройке конвейера интеграции и поставки (мы не будем рассматривать этот процесс подробно, это отдельная тема, которой обычно занимаются отдельные специалисты, так называемые девóпсы, инженеры по внедрению)
- docs (понятно) – обвязка кода комментариями, добавление поставляемых отдельно справочных файлов и т.д.
- feat (от «feature» – «функционал», «фича») – добавление новой фичи
- fix (понятно) – исправление ошибки
- perf (от «performance» – «производительность») – улучшение производительности, уменьшение потребления ресурсов и т.д.
- refactor «переделка», «рефакторинг») – рефакторинг кода (ликвидация технического долга, улучшение структуры кода и т.д.)
- style (понятно) – правки по стилю (форматирование отступов, именование переменных и т.д.)
- test (понятно) – обвязка модульными (а может, и другими) тестами.

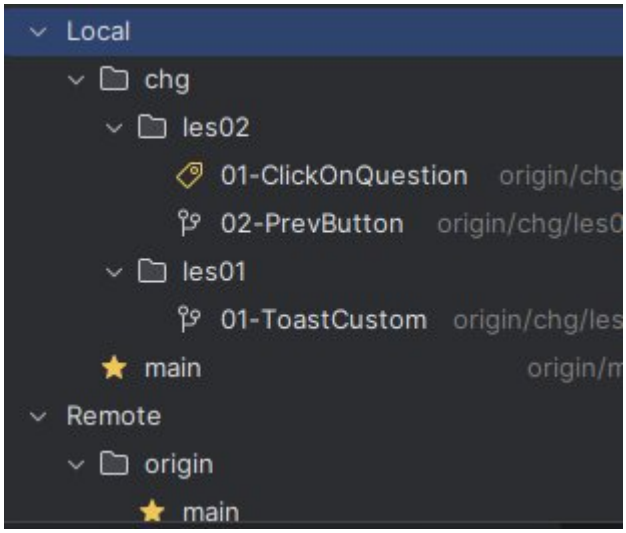
Допустим, нам в подпроект с микросервисом работы с базой пользователей нашего проекта пришёл отчёт об ошибке, позволяющей зарегистрировать невалидный e-mail. На основании отчёта об ошибке в трекере задач была поставлена задача номер 314. Тогда ветка для исправления данной ошибки могла бы называться, например, так:

```
fix/USER-314-disallow-invalid-emails
```

Таким образом, при взгляде на название ветки сразу понятно и какого рода задача решается, и где задачу искать в трекере задач, и в чём, в общих чертах, задача состоит.

Иерархическое именование веток, где сначала идёт тип предназначения ветки, затем слэш, затем название (или, быть может, подтип, затем опять слэш и уже название) имеет ещё и тот плюс, что некоторые графические клиенты Git распознают такую иерархию и соответствующим образом отображают перечень

существующих веток. Вот, например, как может выглядеть набор веток в графическом клиенте Git, встроенном в Android Studio:



Для сообщений коммитов тоже обычно используются те или иные соглашения. В самом простом виде, когда описать коммит можно одной строчкой, для этой строчки оптимальным соглашением будет примерно такое:

```
тип: задача – краткое описание
```

где «тип» и «задача» имеют то же значение, что и для веток (однако, сокращённые слова типа «feat» во многих командах при написании сообщений коммитов меняют на полные - «feature» вместо «feat», «performance» вместо «perf» и т.д.)

Также популярно соглашение, когда для коммитов после типа в скобках указывается буквально одним словом область, где произведено изменение. Скажем, если для исправления ошибки в примере выше потребовалось внести изменения в валидатор e-mail, логично будет написать сообщение, например, так:

```
fix (validator): USER-314 – корневой домен теперь не короче двух букв
```

Да, информация в имени ветки и в сообщении коммита частично дублируется. Это норма – после принятия запроса на слияние ветка может быть удалена, а сообщение коммита останется в истории.

Если имеет смысл описать изменения, зафиксированные коммитом, поподробнее, лучше создать многострочное сообщение. При пользовании Git с командной строки (в том числе через Git Bash) мы при этом просто не пишем сообщение в самой команде git commit – в этом случае Git откроет редактор для написания сообщения (по умолчанию обычно Vim, но может быть настроен и другой). При использовании средств, предоставляемых средой разработки, обычно есть возможность ввести многострочное сообщение в самой среде.

Многострочное описание не отменяет краткого – при просмотре истории коммитов в сокращённом виде, когда на каждый коммит отводится по одной строке, вы увидите именно краткое описание.

Обычно между кратким и многострочным описанием в сообщении коммита отбивают одну пустую строку, так читается легче.

Отдельным случаем являются так называемые WIP-коммиты. Это сокращение от «work in progress». Действительно, так называют коммиты, которые нарушают принцип «одно законченное атомарное изменение – один коммит». То есть, WIP-коммит – это коммит, код в котором может быть неконсистентным, может даже не компилироваться. Единственное предназначение такого коммита – срочное сохранение работы на текущий момент. При грамотной организации работы вам, скорее всего, не понадобится создавать WIP-коммиты, но ситуации бывают разные (и большинство из ситуаций, требующих создания такого коммита, далеки от понятия «всё хорошо»). Например, в офисе может быть пожар – сотрудник быстро отправляет WIP-коммит в ветку, в которой работает, чтобы, если компьютер выйдет из строя, после ликвидации пожара продолжить работу на другом. Сотрудник может почувствовать себя плохо и быть вынужденным уехать в больницу – чтобы коллега мог взять на себя его задачу, а уже сделанное не пришлось делать заново, вполне подойдёт WIP-коммит.

Сообщение WIP-коммита вполне естественно начинать со строчки «WIP» (а иногда ей и заканчивать, если счёт идёт на секунды).

НИКОГДА не следует отправлять WIP-коммиты в основную ветку. Это, кстати, ещё один аргумент в пользу того, чтобы всегда, для любых изменений создавать отдельные ветки, которые затем вливаются в основную.

Хорошо, допустим, мы начали работу над задачей в отдельной ветке, возникла нештатная ситуация, мы создали WIP-коммит, затем, после разрешения нештатной ситуации, вернулись к работе, завершили работу над задачей и создали уже нормальный коммит. Как это вливать в основную ветку? Если просто, ничтоже сумняшеся, создать запрос на слияние, то все коммиты сливаемой ветки попадут в основную, включая неконсистентный WIP-коммит. Это определённо не то, что мы хотим. Нам бы как-то из этих двух коммитов сделать один нормальный.

Для этого служит интерактивное перебазирование, или перебазирование в диалоговом режиме. Чтобы его запустить, нужно команду git rebase сделать с ключом -i и указать ей, история скольких коммитов, начиная от текущего (HEAD) должна быть перебазирована. Для этого указывается параметр HEAD~n, где n – количество коммитов.

Интерактивное перебазирование – действие довольно сложное, поэтому давайте рассмотрим его на примере. Возьмите репозиторий, с которым мы уже экспериментировали – MyAwesomeCoolSite. Если вы не в ветке master, то перейдите в неё:

```
git switch master
```

Отпочкуйте от последнего коммита ветку с названием, допустим, tutor/wip-example, и сразу перейдите в неё:

```
git switch -c tutor/wip-example
```

Допустим, мы начали работать и были вынуждены выполнить WIP-коммит. Начните что-то писать в файле index.html, сохраните, выйдите. Сделайте коммит с сообщением «WIP: пример прерванной работы» и запушьте его в облако:

```
git commit -a -m "WIP: пример прерванной работы"

git push -u origin tutor/wip-example
```

Напоминаем: флаг -a добавляет в коммит только уже известные Git измененные файлы. Если вы создали новые файлы, они не попадут в коммит. Поэтому надежнее сначала выполнить git add . (или git add имя_файла), а затем git commit -m "..." без флага -a. Но в данном конкретном случае у нас index.html уже был закоммичен ранее, поэтому мы и использовали -a.

Теперь допустим, мы смогли вернуться к прерванной работе. Опять откройте файл index.html в редакторе, напишите ещё что-нибудь и снова сохраните. Сделайте коммит с сообщением «tutor: работа продолжена» и запушьте:

```
git commit -a -m "tutor: работа продолжена"

git push
```

Во втором случае можно пользоваться командой git push без параметров, потому что связь с новой веткой уже установлена.

Посмотрим на историю коммитов в кратком формате, когда на каждый коммит отводится по одной строке:

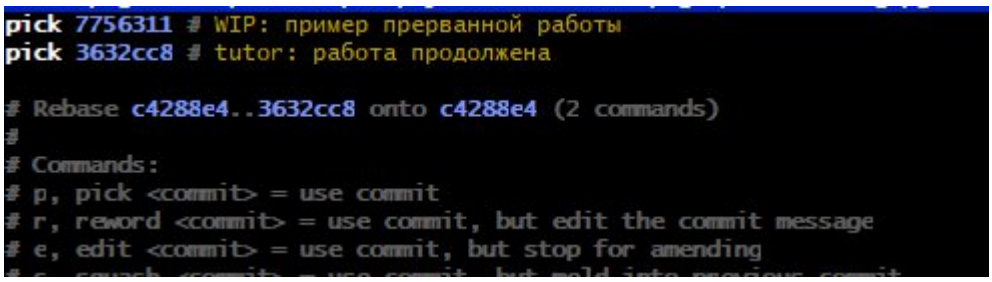
```
git log --oneline
```

Мы видим, что в нашей ветке присутствует WIP-коммит. Это, конечно, очевидно, но перед тем, как создавать запрос на слияние, мы бы хотели слить его с нормальным коммитом – чтобы не было неконсистентных изменений и сообщения WIP, а был один нормальный коммит, который содержал в себе все сделанные изменения.

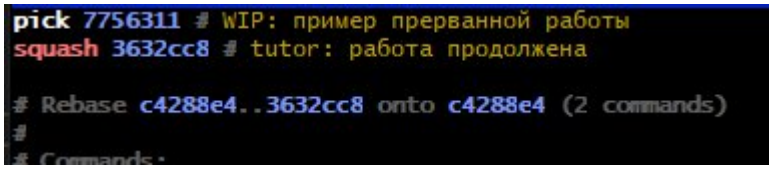
Начнём перебазирование двух последних коммитов в диалоговом режиме:

```
git rebase -i HEAD~2
```

Мы видим, что Git открыл редактор (Vim, Nano или другой), где указаны два последних коммита (сначала WIP, затем нормальный), перед которыми стоит слово «pick». Ниже даны комментарии, которые рассказывают, какие команды можно применять для коммитов в этом режиме.

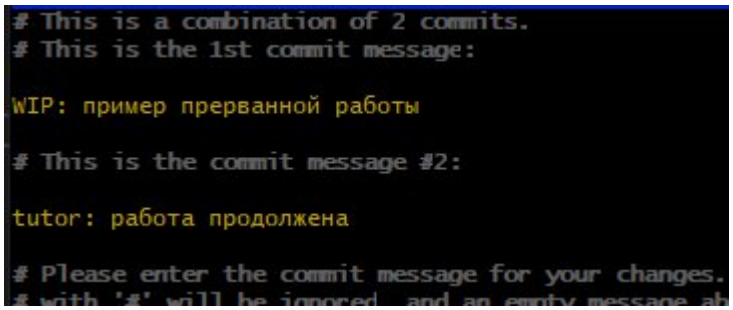


Команда pick означает, что мы хотим оставить этот коммит в истории. Команда reword означает, что коммит мы хотим оставить, но дать ему другое сообщение. Команда edit означает, что этот коммит мы хотим отредактировать (добавить какие-то изменения, удалить и т.д.). Команда drop означает, что мы хотим убрать из истории этот коммит – эта команда, на самом деле, нам **НЕ** подходит, поскольку нам надо сохранить внесённые коммитом изменения, нам только не надо, чтобы они были отдельным коммитом. Та команда, которая нам нужна – это команда squash. Она соединяет тот коммит, которому дана эта команда, с **ПРЕДЫДУЩИМ**. Поскольку коммит объединяется с предыдущим, а не с последующим, нам нужно применить команду squash к тому коммиту, который идёт **ПОСЛЕ** нашего WIP-коммита. Вносим соответствующие изменения прямо в редакторе:

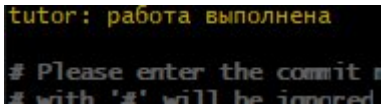


и выходим с сохранением.

Git опять открывает редактор, но уже для того, чтобы мы придумали сообщение для коммита, заменившего собой те два, которые мы объединили командой squash:



Мы видим, что Git сохранил для нас сообщения обоих коммитов, причём сначала идёт сообщение WIP. В нашем конкретном случае удалим их оба и введём сообщение, например, «tutor: работа выполнена» (в настоящем проекте за такое сообщение коммита могут поступить с его автором очень грубо, но мы сейчас просто изучаем то, как избавляться от WIP в истории):



Сохраним файл и выйдем. Если мы теперь опять посмотрим на историю, то увидим вместо двух коммитов, один из которых WIP, один нормальный коммит, содержащий в себе изменения из обоих коммитов и сообщение без пометки «WIP».

Чтобы подать запрос на слияние, нам необходимо отправить сделанные изменения в облако. Но если мы попробуем это сделать, ничего не получится. Это из-за того, что теперь наша локальная история перестала соответствовать истории в облаке.

По счастью, мы работаем в отдельной ветке, в которой никто, кроме нас, не работает. Поэтому мы можем выполнить принудительную отправку:

```
git push --force
```

Если бы все работали в основной ветке, мы бы этой командой осложнили жизнь всем коллегам. Так что это ещё один аргумент в пользу создания отдельной ветки под каждую задачу. Кстати, есть более безопасный вариант – флаг --force-with-lease (буквально – «силой, но с оговоркой»). С таким флагом также будет выполняться принудительная отправка, но в случае, если наша локальная копия ветки не самая свежая (что означает, фактически, что туда успел спушиться кто-то ещё), отправка не будет произведена. Иными словами, Git проверит, не изменилась ли удалённая ветка с момента нашего последнего обращения к ней, и не даст испортить работу коллеги, если вдруг он туда что-то успел спушить. При использовании этого флага проблемы в случае такой ситуации возникнут не у всей команды, а только у нас. Что, впрочем, тоже малоприятно, хотя и далеко не так катастрофично, как в предыдущем случае.

Заметим, что объединение WIP-коммитов с нормальными консистентными коммитами через squash в интерактивном перебазировании – это общий случай. Он подходит, когда у нас в ветке много коммитов, и нам нужно объединить только несколько из них (скажем, у нас пять коммитов, из них один WIP). Если нам достаточно всю ветку объединить в один коммит, есть способ проще. Здесь нам опять поможет команда git reset. Мы ранее использовали её для того, чтобы вынести из индекса добавленный туда файл (с именем файла в качестве параметра и без флагов) и для того, чтобы удалить сделанные изменения (с флагом --hard). Теперь мы рассмотрим ещё один вариант – когда эта команда отменяет сделанные изменения вплоть до какого-то момента, но не удаляет их, а оставляет в рабочей копии файлов. То есть все изменения есть, но они как будто бы не были ни закоммичены, ни даже добавлены в индекс. Для этого в качестве параметра используется указание на последний из коммитов, который должен остаться нетронутым, и флаг --soft. Например, если нам нужно отменить два коммита, но оставить изменения в незакоммиченном состоянии, мы можем выполнить команду

```
git reset --soft HEAD~2
```

В случае, когда мы хотим всю ветку объединить одним коммитом, мы можем сначала отменить (оставив в индексе) все выполненные в ней коммиты, то есть сбросить её до того момента, где она отпочковалась от ветки master:

```
git reset --soft master
```

а затем выполнить коммит обычным образом и с нужным сообщением. Конечно, если коммиты ветки пушились в облако, результат нужно будет пушить опять же с флагом --force (или --force-with-lease). Однако, возможно, безопаснее будет делать такой откат не к master, а к HEAD~n, где n – количество коммитов в вашей ветке. Так вы гарантированно отмените коммиты только в пределах вашей ветки.

Вопросы для самопроверки:

1. Выберите причины (может быть несколько), по которым не стоит вести разработку непосредственно в основной ветке.

   Варианты ответов:

   1.1. Потому что из-за длины основной ветки коммиты в неё производятся долго.

   1.2. Потому что код в основной ветке должен быть всегда готов к поставке.

   1.3. Потому что облачные репозитории позволяют вносить изменения в основную ветку только через запросы на слияние.

   1.4. Потому что при необходимости выполнить WIP-коммит его будет сложно устранить из основной ветки.

2. Как лучше создать ветку для разработки новой фичи?

   Варианты ответов:

   2.1. Непосредственно из текущей ветки.

   2.2. Из основной ветки (обычно master или main).

   2.3. Не следует создавать ветку для новой фичи, если она не приводит к увеличению старшего номера версии продукта.

§ 2.5. Поставка кода

Чаще всего разработчик поставляет конечный продукт. Если это веб-приложение, то оно разворачивается на сервере, если это десктопное приложение, то оно скачивается с сайта или предоставляется через репозиторий приложений, если это мобильное приложение, то оно практически наверняка предоставляется через репозиторий приложений (или магазин приложений, как ещё говорят, подразумевая возможность платной поставки).

Однако иногда бывает, что разработчик должен поставить не только конечный продукт, но и исходный код, из которого этот продукт собирается. Это может быть при разработке продукта на заказ, при разработке по модели аутсорсинга (когда разработчик выступает фактически субподрядчиком своего заказчика). Если разрабатываемый продукт имеет определённые требования к безопасности (это в первую очередь госсектор, банкинг и интернет-торговля), может возникнуть также необходимость предоставить исходный код ответственным за анализ уязвимостей продукта.

Во многих случаях разработчик не испытывает желания и не имеет необходимости предоставлять доступ к репозиторию, со всеми его ветками, историей коммитов, правами на запись. Всё, что нужно – это собрать архив с кодом и отправить его. Как это сделать?

Собирать архив с исходным кодом из рабочей папки, как правило, нерационально – в ней имеются исполняемые файлы, устанавливаемые модули, секретные настройки... в общем, всё то, что мы обычно добавляем в список игнорирования.

Простейший вариант, который напрашивается – это создать новую папку, склонировать туда репозиторий, потом удалить папку .git и всё остальное заархивировать. Многие так и делают. Однако, Git, тем не менее, предоставляет отдельную команду, которая позволяет всё это осуществить без создания новой папки и к тому же за один вызов – это команда git archive.

При помощи команды git archive можно создать архив в формате ZIP, TAR либо TGZ (возможно, список будет пополняться) из любого зафиксированного в Git состояния репозитория, как локального, так и удалённого, причём опционально можно отказаться от включения в архив каких-либо файлов.

В самом простом случае, когда нам надо просто получить архив текущего коммита, мы можем выполнить эту команду следующим образом:

```
git archive –o имяархива.zip HEAD
```

Здесь имяархива.zip – это имя, которое получит файл архива, оно указывается после ключа -o от слова «output» («выход»). HEAD, как мы уже упоминали, представляет собой ссылку на текущий коммит.

Обратите внимание: Git определяет формат создаваемого архива (ZIP, TAR и т.д.) **автоматически, по расширению файла**, которое вы указываете после ключа -o. Поэтому важно писать .zip или .tar корректно.

Если нам нужно получить архив последнего коммита конкретной ветки, мы можем указать вместо HEAD имя ветки, а если нам нужно получить архив некоего заданного коммита, мы можем указать его краткий хэш-код (можно, конечно, и полный).

Если мы хотим получить архив какого-то коммита из облачного репозитория, мы можем сказать об этом при помощи ключа --remote, после которого нужно будет указать имя, под которым мы зарегистрировали облачный сервис (origin, upstream и т.д.), в остальном создание архива из облачного коммита ничем не отличается от создания архива из локального коммита (но, конечно, нужно будет ввести пароль от закрытого ключа, если ключ защищён).

Наконец, мы можем захотеть исключить из архива какие-то файлы. Для этого мы можем использовать ключ --exclude. Например, если в репозиторий фронтенда веб-приложения входит файл конфигурации веб-сервера под именем, скажем, nginx.default.conf, а предоставлять его в архиве мы не хотим (например, архив создаётся для передачи в отдел проверки уязвимостей, и им этот файл, допустим, не нужен), мы можем создать архив следующим образом:

```
git archive –o frontend.zip ––exclude nginx.default.conf HEAD
```

Пожалуй, главное преимущество git archive перед обычным созданием ZIP-архива через Проводник или стороннюю программу (например, WinRAR или 7-Zip) заключается именно в том, что эта команда **автоматически и строго соблюдает правила вашего файла .gitignore**. Она гарантированно не включит в архив никакие неотслеживаемые (untracked) файлы, временные сборки или секреты, даже если они физически лежат в папке проекта. Она архивирует только то, что находится под контролем версий.

Конечно, говоря о поставке кода, следует упомянуть и поставку самого приложения. Обычно это делается автоматически при помощи настроенных девопсами конвейеров CI/CD. Но нужно иметь в виду два нюанса: во-первых, есть смысл в журнале коммитов помечать более или менее знаковые релизы, а во-вторых, сами конвейеры иногда имеет смысл настраивать так, чтобы поставлять не всё подряд, а только то, что прошло одобрение руководителя проекта как заслуживающее поставки. Обе эти задачи решает механизм тегов, или меток.

Теги – это просто именованные указатели на тот или иной коммит, как ветки, но отличается принцип использования. Ветки мы создаём, сливаем, перебазируем и т.д. Указатель на «голову» ветки используется для прохождения по всей истории коммитов ветки. Теги же служат для указания просто на какой-то конкретный коммит, чтобы его можно было быстро найти. В отличие от веток, которые постоянно сдвигаются при новых коммитах, теги неизменяемы. Они «приклеиваются» к конкретному коммиту навсегда. Это делает их идеальными

маркерами релизов: история версии 3.14 всегда будет указывать на тот же снимок кода, что и в день выпуска.

В Git есть два основных типа тегов: легковесные и аннотированные. Легковесный тег содержит просто ссылку на конкретный коммит и имя тега. Например, если мы очередной коммит внедрили как версию нашего продукта 3.14, имеет смысл как-то указать в журнале коммитов, что это коммит, соответствующий версии 3.14. Естественный способ: создать метку с текстом, скажем, «v3.14». Для этого используется команда

```
git tag v3.14
```

На практике теги обычно именуют по стандарту SemVer: v1.0.0, v1.2.3 и т.д. Префикс v не обязателен, но широко принят, чтобы отличать версию программы от просто числа.

Зачастую девопс настраивает конвейер поставки в продуктовую среду так, чтобы он запускался только при появлении тега определённого вида (например, «v*.*.* stable»). Это, с одной стороны, исключает случайную поставку в продуктовую среду недостаточно проверенного кода, а с другой стороны, делает поставку после её утверждения максимально простой и прозрачной как на момент поставки, так и на протяжении дальнейшей истории.

Аннотированный тег, кроме этого, содержит сообщение (помимо сообщения самого коммита) и прочую информацию. Для этого используется команда примерно такого вида:

```
git tag –a v3.14 –m "Релиз 3.14 от 2024–05–18"
```

Здесь флаг -a служит для указания, что создать надо именно аннотированный тег, а -m указывает, что дальше идёт сообщение тега.

Посмотреть все теги репозитория можно командой tag без параметров (как в случае branch для просмотра веток):

```
git tag
```

Можно также выполнить поиск по шаблону. Например, чтобы найти теги, соответствующие (в нашем примере) всем релизам с ведущим номером 3, можно выполнить команду (от list):

```
git tag –l "3.*"
```

Если тег больше не нужен, его можно удалить (от delete):

```
git tag –d "3.14"
```

Команда push по умолчанию не отправляет в облако информацию о тегах. Чтобы эта информация отправилась, используется специальный флаг:

```
git push ––tags
```

Однако такой вариант команды отправит в облако вообще все локальные теги данного репозитория, включая черновые, тестовые, экспериментальные и прочие, созданные только для себя. Чтобы отправить только аннотированные теги, используйте команду

```
git push --follow-tags
```

Однако ещё лучше отправлять теги по мере их создания и только в том случае, когда тег действительно предназначен для всей команды. Для этого отправляемый тег указывается явно:

```
git push origin v3.14
```

Есть и другие флаги для работы с тегами, мы перечислили только основные.

Часто архив делают не по ветке или хэшу, а именно по тегу:

```
git archive -o release_v3.14.zip v3.14
```

Это связано в том числе с тем, что теги обычно маркируют стабильные версии, готовые к передаче заказчику.

Вопросы для самопроверки:

1. Какой вариант кода мы НЕ можем поставить при помощи команды git archive?

Варианты ответов:

1.1. Вариант, зафиксированный в ветке, отличной от основной.

1.2. Рабочую версию, если она отличается от закоммиченной.

1.3. Старый коммит.

1.4. Коммит, присутствующий в облаке, но отсутствующий локально.

§ 2.6. Что делать и кто виноват

Немаловажная часть рабочего процесса в разработке – это поиск возникающих ошибок. Хорошо, когда новая ошибка заметна сразу – в этом случае есть довольно большая вероятность, что достаточно быстро будет найдено то конкретное изменение кода, которое повлекло за собой нежелательное поведение приложения. К сожалению, иногда бывает, что ошибка весьма нетривиальна и средствами автоматического тестирования не обнаруживается. Например, бывает так, что ошибка проявляется лишь при особом сочетании факторов, которое не сразу пришло в голову тестировщикам. Такие ошибки иногда выявляются лишь в процессе эксплуатации. При этом между двумя версиями, поступившими в эксплуатацию, могут быть десятки коммитов, и уже не так легко определить, в какой момент закралась ошибка. Нам помогут приёмы работы с журналом коммитов.

Как мы уже упоминали ранее, для просмотра журнала коммитов служит команда

```
git log
```

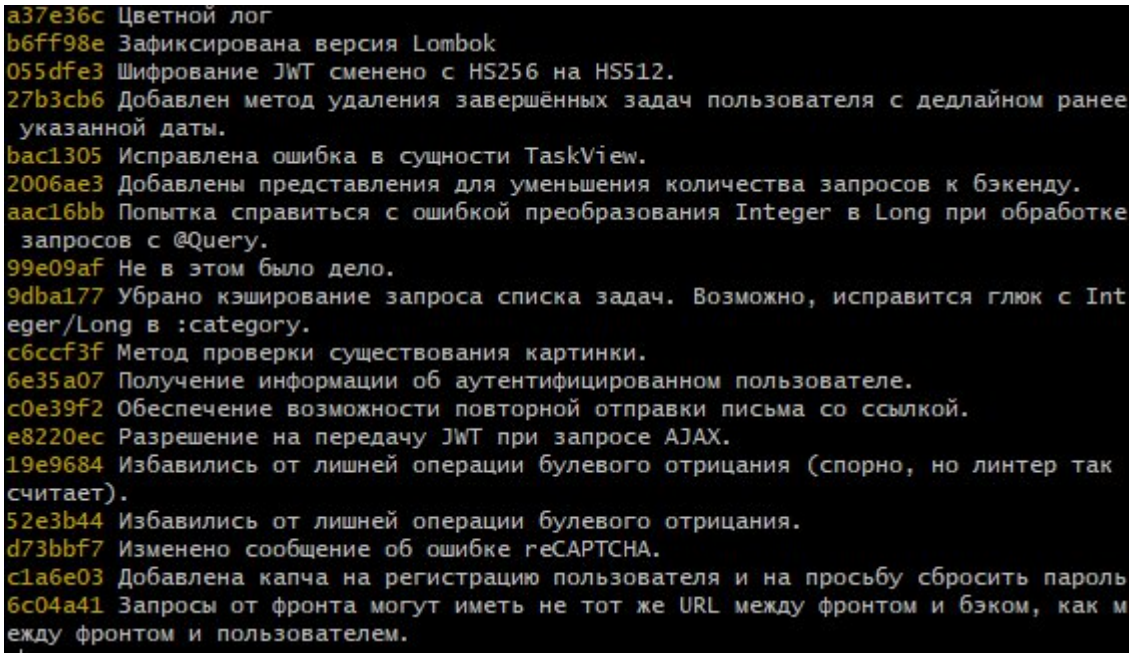
По умолчанию эта команда выводит довольно много информации о каждом из коммитов:

```
commit 0ae5852a204f7c0e4a6a14323ecaa2f99ff6d46f
Author: Гершман Антон Лейбович <allegecityrat@yandex.ru>
Date:   Thu Oct 16 23:41:35 2025 +0300

    feat: удаление записи

commit 21ad398eafaf1720599e6111dc5ecfa5204b7e6c
Author: Гершман Антон Лейбович <allegecityrat@yandex.ru>
Date:   Thu Oct 16 23:18:36 2025 +0300

    feat: изменение значения поля записи

commit 612335bf7b7996e363eacb1dd332c4ca65eb46ef
Author: Гершман Антон Лейбович <allegecityrat@yandex.ru>
Date:   Thu Oct 16 22:52:21 2025 +0300

    feat: добавление записи

commit bce5faa8005e651ad97814a50c0493cb5c3bf146
Author: Гершман Антон Лейбович <allegecityrat@yandex.ru>
Date:   Thu Oct 16 22:44:50 2025 +0300
```

Как мы видим, по умолчанию выводится полный хэш каждого коммита, ФИО и e-mail автора, дата и время добавления коммита, а также краткое сообщение о коммите. Когда коммитов много, они не помещаются на экран. В этом случае мы можем перемещаться по истории клавишами со стрелками либо Page Up / Page Down, а также, если мы нашли то, что хотели, можем выйти из режима многоэкранного показа журнала коммитов, нажав на клавиатуре клавишу Q (от «quit»).

Всё это в чём-то, конечно, хорошо, но для целей, когда нам надо одним взглядом окинуть как можно больший кусок истории изменений (если коммитов

было много), такой режим не очень-то подходит. Так, в данном случае мы единовременно видим только четыре коммита. При обширной истории изменений нам этого может даже не хватить для того, чтобы, исходя из логики развития продукта, понять, в какую сторону нам следует двигаться, чтобы найти нужный коммит. Мы можем улучшить эту ситуацию, используя режим, в котором на каждый коммит отводится по одной строке:



Это делается при помощи команды

```
git log --oneline
```

В этом случае на каждый коммит выдаётся минимум информации: краткий хэш и первая строка сообщения (в случае, если коммит является последним в какой-либо ветке, может быть также выведено название ветки). Зато за счёт этого на экране помещается гораздо больше коммитов, точное их количество зависит от размеров экрана (на скриншоте их более двадцати). Конечно, если коммитов было много, то и в этом случае по журналу можно ходить стрелками и клавишами PgUp/PgDn, а выходить из истории клавишей Q.

Если мы хотим в той же строке видеть теги (когда они есть), добавьте флаг decorate:

```
git log --oneline --decorate
```

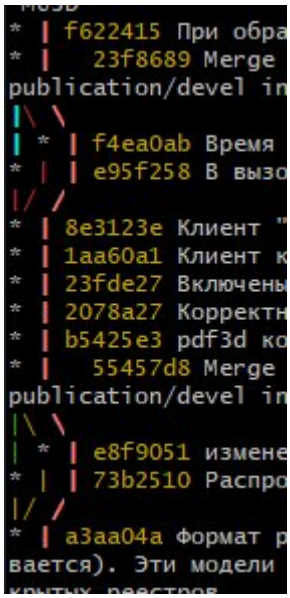
Вы увидите, как рядом с коммитом появится его метка:

```
a1b2c3d (tag: v3.14, origin/main, main) feat: релиз 3.14
```

Можно включить графическое представление, какой коммит к какой ветке относится, и какая ветка откуда почкуется и в какую затем сливается – для этого служит ключ graph всё той же команды git log:

```
git log --oneline --graph
```

Вот как может выглядеть результат (сообщения обрезаны):



Можно выбрать для отображения только коммиты, удовлетворяющие определённым критериям:

```
git log --oneline --author="Basil Pupkind" --after="1 week ago"
```

Вернёмся к поиску ошибок.

Например, исходя из симптомов ошибки, мы понимаем, что ошибка кроется в определённой функции. Мы знаем, что тело функции располагается в файле, допустим, ScoreCounter.cs где-то между строками с номерами 35 и 90 (точное значение может меняться – всё-таки в процессе изменений могли добавляться или удаляться строки как до, так и после нашей функции). Как нам сделать так, чтобы в журнал попали только те коммиты, в которых были изменения в этом файле в этом диапазоне строк? При помощи команды

```
git log -L 35,90:ScoreCounter.cs
```

Здесь L – от слова line (строка). В этом режиме ключ oneline не работает – для каждого найденного коммита Git будет показывать произошедшие в указанном диапазоне строк указанного файла изменения.

Также мы можем пойти от противного – посмотреть, в каком коммите произошло последнее изменение каждой строки указанного диапазона указанного файла. Для этого служит команда git blame (от «to blame» – «обвинять»):

```
git blame -L 35,90 ScoreCounter.cs
```

Обратите внимание – в команде log диапазон строк и имя файла разделялись двоеточием, а в команде blame пробелом.

По этой команде Git выведет для каждой строки указанного диапазона указанного файла краткий хэш, автора, дату и время коммита, а также номер строки и её последнее зафиксированное содержимое. Таким образом, если нам по виду

строки понятно, что она содержит ошибку, то мы можем довольно быстро найти коммит, в котором эта ошибка появилась, и откатить его целиком.

Важно помнить, что команда git blame в психологически здоровом коллективе отвечает не на вопрос «Кто накосячил?», а на вопрос «Почему этот код написан именно так?». Часто, посмотрев автора и дату, мы находим ссылку на задачу в трекере или старый коммит, который объясняет, что это не ошибка, а намеренное (хоть и, может быть, странное) решение, продиктованное старыми требованиями.

Кстати, в большинстве команд (в первую очередь в команде git diff) мы можем указать, чтобы изменения, касающиеся только пробельных символов, не считались за изменения. Для этого к команде нужно добавить ключ -w.

Хорошо, а если у нас нет никаких идей по поводу того, где возникла ошибка, кроме того, что мы знаем, что в пущенной в релиз версии от такого-то числа этой ошибки не было, а в пущенной в релиз версии от сякого-то числа она уже была? Тогда единственный способ понять, в каком коммите она возникла – это проверить код, зафиксированный в каждом из коммитов, и понять, в какой момент всё стало плохо.

На самом деле, не совсем так, и если вы знакомы с концепцией двоичного поиска, вы это поняли уже пока читали предыдущий абзац. Конечно же, нам не нужно проверять каждый коммит. Нам нужно сначала взять тот коммит, который находится посредине между точно содержащим искомую ошибку и точно её содержащим. Если этот коммит содержит ошибку, значит, она появилась не позже, и надо взять коммит между точно не содержащим ошибку и последним взятым коммитом, а если коммит её не содержит, значит, она появилась позже, и надо взять коммит между ним и точно содержащим ошибку. И так далее, каждый раз сужая область поиска примерно вдвое, мы найдём момент появления ошибки достаточно быстро – для истории из тысячи коммитов мы гарантированно найдём тот, в котором возникла ошибка, за десять попыток.

Для того, чтобы временно откатить состояние файлов проекта к моменту некоего известного коммита, используется уже известная нам команда git checkout (мы уже говорили, что у неё много разных функций):

```
git checkout aaaaa
```

где aaaaa – краткий (обычно вполне достаточно четырёх-пяти первых символов) хэш коммита. При этом репозиторий переходит в особое состояние «detached HEAD» (отсоединённая «голова»), когда мы не находимся ни в одной ветке. Если об этом забыть и случайно сделать коммит, этот коммит также не будет принадлежать ни одной ветке и будет потерян, как только мы вернёмся в ту или иную ветку.

Чтобы не забыть об этом нюансе, лучше использовать команду switch с явным указанием флага detach:

```
git switch --detach aaaaa
```

Этот флаг явно переводит репозиторий в режим «detached HEAD», предупреждая вас о том, что вы просматриваете историю, а не работаете в ветке. В этом режиме можно смотреть код и запускать тесты, но не следует делать новые коммиты, так как они могут быть потеряны при переключении обратно. Чтобы вернуться в свою ветку, просто выполните git switch master (или имя вашей ветки).

Вместе с тем, если у нас, действительно, пусть даже не тысяча, пусть даже тридцать-сорок коммитов, производить даже двоичный поиск между ними вручную непросто. Придётся либо через раз выполнять заново команду git log с нужными параметрами, чтобы смотреть хэш-коды нужных коммитов, либо выписать (лучше, конечно, распечатать) их заранее. Неужели нельзя это как-то автоматизировать?

Конечно же, можно. Для этого служит команда git bisect с разнообразными параметрами.

Для начала нужно запустить режим двоичного поиска командой

```
git bisect start
```

После этого нужно указать Git последний коммит из тех, о которых мы можем точно сказать, что в нём ошибки ещё нет, и первый коммит из тех, о которых мы можем точно сказать, что ошибка уже есть. Для этого служат команды git bisect good и git bisect bad, сопровождаемые кратким хэшем указываемого коммита. Например:

```
git bisect good 73b25
```

и

```
git bisect bad f6224
```

Сразу после этого Git самостоятельно найдёт коммит посередине между ними, выполнит переход на него и сообщит, что можно проверить полученное состояние проекта. Мы проверяем и выполняем команду

```
git bisect good
```

если это состояние ещё не содержит искомой ошибки или

```
git bisect bad
```

если содержит. Далее Git самостоятельно найдёт следующий коммит, который необходимо проверить, и выполнит переход на него. И так мы будем продвигаться к цели, каждый раз оказываясь к ней вдвое ближе, чем до того, пока Git не укажет нам точно тот самый коммит, где ошибка, собственно, появилась. После этого работа режима двоичного поиска будет завершена.

Если в процессе двоичного поиска вы запутались, передумали искать ошибку или просто хотите вернуться к своей обычной работе, не дожидаясь финала, используйте команду:

```
git bisect reset
```

Эта команда безопасно завершает режим поиска и возвращает вас в ту ветку и на тот коммит, где вы находились до запуска git bisect start.

НА ЗАМЕТКУ:

Если у вас есть автоматический тест (скрипт), который возвращает код 0 при успехе и не-0 при ошибке, вы можете полностью автоматизировать поиск:

```
git bisect run имя_скрипта.sh
```

(здесь предполагается, что мы в Linux)

Git сам будет переключать коммиты, запускать скрипт и определять «good» или «bad», пока не найдет виновника.

Напоследок рассмотрим такую возможность, как сокращение команд. Она впрямую не относится к теме поиска ошибок, но раз уж мы затронули команды с длинными наборами разнообразных ключей, будет, наверное, правильно рассказать, как их сократить. Это делается через настройку Git при помощи команды git config. К любой команде с заданным набором ключей можно определить настройку, задающую её псевдоним – краткую команду, которая будет «видна» только там же, где соответствующая настройка (как мы помним, по области применимости у нас настройки бывают локальные, глобальные и системные), и будет эквивалентна указанной команде с указанным набором ключей. Эти настройки объединяются в группе настроек alias (от английского «alias» – «псевдоним»).

Допустим, нам очень полюбилась возможность просмотра журнала команд в однострочном режиме с графическим отображением веток, но каждый раз вводить

```
git log --oneline --graph
```

нас утомляет. Допустим, мы хотим, чтобы для нас (то есть, как глобальная настройка) существовала команда, скажем, git glogg, которая была бы эквивалентна команде git log с ключами oneline и graph. Мы создаём в группе alias настройку с именем glogg, значение которой – требуемая нам команда (поскольку в ней содержатся пробелы, необходимо заключить её в кавычки):

```
git config --global alias.glogg "log --oneline --graph"
```

Теперь при выполнении команды

```
git glogg
```

фактически будет выполняться команда

```
git log --oneline --graph
```

Если нам по какой-то причине ранее созданный псевдоним больше не нужен, мы можем убрать его из настроек при помощи флага unset:

```
git config --global --unset alias.glogg
```

Вопросы для самопроверки:

1. Для чего делается команда git blame?

   Варианты ответов:

   1.1. Чтобы лишить права коммитить того, кто создал ошибочный коммит.

   1.2. Чтобы для каждой строки кода увидеть, кто и когда сделал коммит с ней.

   1.3. Чтобы найти, в каком коммите файл был создан.

   1.4. Чтобы автоматически исправить ошибки в указанном файле.

2. Какой ключ команды git log обеспечивает краткий формат вывода журнала?

   Варианты ответов:

   2.1. --brief

   2.2. --fast

   2.3. --oneline

   2.4. --online

Часть 3. Закрепляем на практике

В этой части мы выполним небольшую лабораторную работу для того, чтобы закрепить на практике изученный материал.

§ 3.1. Создание репозитория

Откройте Git Bash (если вы работаете в Windows) или просто терминал (если работаете в Linux). При помощи команды cd перейдите в папку, в которой хотите выполнять лабораторную работу.

Представим себе, что мы создаём репозиторий для работы над проектом редактора компьютерных игр с открытым миром. Там можно будет создавать игровые миры (realms), придумывать разные сущности (entities) с теми или иными свойствами (properties) и расставлять в игровом мире объекты (objects), представляющие собой экземпляры этих сущностей (например, мы придумываем сущность «дерево» со свойствами «высота», «обхват ствола» и т.д., и наполняем игровой мир различными деревьями).

Но сейчас наша задача самая что ни на есть скромная – мы создаём репозиторий для работы над этим проектом и пишем документацию по функциям этого редактора.

Создайте новую папку с названием realms_editor. Напомним, что для создания папки используется команда mkdir.

Чтобы превратить созданную папку в репозиторий, надо сначала перейти в неё (используйте команду cd), а затем выполнить команду git init. С этого момента Git сможет отслеживать то, что происходит в данной папке. Свою служебную информацию Git хранит в скрытой папке .git – выполните команду ls с ключом -a, чтобы посмотреть список содержимого папки realms_editor, включая скрытые файлы и папки, и убедитесь, что папка .git появилась.

Убедимся, что репозиторий создан, но в нём пока нет ни одного файла и не выполнено ни одного коммита. Используйте для этого команду git status.

Представим себе ситуацию, что нам запретили использовать «расистское» название «master» для основной ветки, и предписали использовать для неё название «main». Тогда нам надо явным образом создать ветку с таким названием (если мы не автоматизировали это при помощи git config). За одну команду создайте ветку main и перейдите на неё (используйте команду git checkout с ключом -b либо git switch с ключом -c). Убедитесь, что вы создали ветку и перешли в неё, при помощи команды git status.

Наш чистый репозиторий готов, мы находимся в главной ветке и готовы добавлять первые файлы и делать первые коммиты.

§ 3.2. Первые коммиты

По доброй традиции, первый файл, который мы создаём в новом репозитории – это файл README.md, который содержит текстовое описание проекта в формате Markdown. Чтобы создать пустой файл, не открывая его на редактирование, в Linux (а значит, и в Git Bash) служит команда touch. Создайте с её помощью файл README.md:

```
touch README.md
```

Убедитесь (при помощи команды ls или даже хоть просто в Проводнике), что файл создался, откройте его любым текстовым редактором (позволяющим вводить просто текст – т.е. не Word или Write, а Блокнотом, Notepad++ или даже хоть Vim) и напишите в файле заголовок:

```
# Команды редактора игровых миров
```

Значок # в формате Markdown обозначает заголовок первого уровня.

Сохраните изменения в файле и убедитесь, что Git обнаружил новый файл в состоянии «неотслеженный», при помощи git status.

Добавьте файл к индексу при помощи команды git add (вспомните, как это делается). Убедитесь, что внесённые изменения теперь отслежены, при помощи команды git status.

Добавим ещё один файл, в котором, собственно, будем писать документацию по командам редактора.

При помощи команды touch создайте файл editor_api.md. Убедитесь, что сейчас Git видит как отслеженные изменения в виде нового файла README.md, так и неотслеженные в виде нового файла editor_api.md. Добавьте editor_api.md к индексу и посмотрите, как теперь выглядит состояние репозитория.

Мы готовы выполнить первый коммит в этом репозитории. Выполните коммит с сообщением «Начало работы». Убедитесь, что в индексе теперь пусто (это естественно – все изменения, которые были в индексе, ушли в коммит). Теперь репозиторий находится в «чистом» состоянии – то есть, в нём отсутствуют изменения (кроме, быть может, игнорируемых – но их у нас пока и нет) по сравнению с последним коммитом.

Убедитесь, что коммит появился в журнале изменений, при помощи команды git log. Найдите в выданной информации сообщение коммита, своё имя, email, дату коммита и его хэш.

Начнём писать документацию. Она, конечно, будет намного менее подробной и намного менее строгой, чем выглядела бы документация в реальном проекте – ведь мы сейчас изучаем работу с Git, а не правила документирования проектов. Откройте файл editor_api.md в любом текстовом редакторе, позволяющем работать с текстом без форматирования, и введите строки:

```
# Игровой мир

Создание мира – realm.create
```

Мы создали заголовок «Игровой мир», под которым будем записывать документацию по функциям, касающимся работы с миром как таковым, и под этим заголовком разместили документацию по функции создания нового игрового мира.

Сохраните изменения. Проверьте состояние репозитория (в реальности вряд ли понадобится так часто это делать, но мы учимся, так что проверка состояния «после каждого чиха» служит именно этой цели). Мы видим, что опять имеется не добавленное в индекс (not staged for commit) изменение, но файл с документацией теперь обозначен не как new (новый), а как modified (изменённый) – Git видит, что этот файл уже коммитился ранее.

Просмотрите сделанные изменения при помощи команды git diff. Эта команда показывает разницу между рабочей папкой и индексом. Мы видим, что добавленные в файл строки помечены значком + (а других у нас пока и нет).

Добавьте изменения к индексу, проверьте состояние репозитория и выполните коммит с сообщением «feat: создание игрового мира». Создание игрового мира – это определённо «фича», поэтому сообщение мы и начали с префикса «feat:».

Посмотрите журнал изменений. Теперь там два коммита, причём самый новый коммит показан первым.

Добавьте в файл с документацией строку

```
Удаление мира – realm.delete
```

Сохраните изменения, проверьте состояние репозитория, посмотрите сделанные изменения, добавьте их к индексу и закоммитьте с сообщением «feat: удаление игрового мира». Посмотрите журнал изменений. Если он перестал помещаться на одном экране, прокрутите его клавишами со стрелками до конца, а при появлении сообщения «(END)» («конец») нажмите клавишу «Q» (от «quit» - «завершить»).

§ 3.3. Ветки

До настоящего момента мы выполняли коммит непосредственно в основную ветку. По-хорошему, так делать не надо – мы уже обсуждали причины. Так что далее мы будем для каждого имеющего определённый смысл изменения создавать отдельную ветку.

Посмотрите список имеющихся веток (команда git branch). Вы видите только ветку «main». Возле неё стоит звёздочка, что означает, что мы сейчас находимся в этой ветке.

Создадим ветку для документирования создания новой сущности – назовём ветку «feat/entity-create» (будем сразу следовать соглашению, по которому ветки для новых «фич» мы называем с префиксом «feat/», а ветки для исправления ошибок – с префиксом «fix/»). Используйте для этого команду git branch без ключей и с именем создаваемой ветки.

После создания новая ветка содержит ровно те же файлы ровно с тем же содержимым и ровно с той же историей коммитов, что и ветка, в которой мы были, когда создавали новую ветку (в данном случае «main»).

Посмотрите список веток теперь. Мы видим, что новая ветка появилась, но текущая ветка до сих пор main. Перейдите в созданную ветку при помощи команды git checkout или git switch. Убедитесь, что переход произошёл, при помощи списка веток.

Добавьте в конец файла документации строки:

```
# Сущность

Создание сущности – entity.create
```

Сохраните изменения, проверьте состояние репозитория, посмотрите сделанные изменения, добавьте их к индексу и выполните коммит с сообщением «feat: создание сущности».

Посмотрите журнал изменений. Он уже практически наверняка вышел за границы экрана, и уж в любом случае его теперь не очень легко охватывать взглядом. Чтобы показывать только самое необходимое, используйте ключ oneline:

```
git log --oneline
```

Обратите внимание – коммит с удалением игрового мира помечен как «(main)», а коммит с созданием сущности – как «(HEAD -> feat/entity-create)». Это как раз указание, что ветка main пока заканчивается на удалении игрового мира, а ветка, в которой мы сейчас находимся, и которая называется feat/entity-create – на создании сущности.

Переключитесь обратно на ветку main. Откройте файл editor_api.md. Мы видим, что строчка с описанием создания сущности исчезла. Всё правильно – в ветке main

ведь её нет. Посмотрите на журнал изменений – соответствующего изменения также не видно, оно ведь в другой ветке. Сейчас у нас в ветке main три коммита, а в ветке feat/entity-create – четыре (три из которых общие с веткой main).

Мы удовлетворены описанием создания сущности (предположим), и готовы влить его в основную ветку, где мы сейчас и находимся. Для этого нам нужно знать имя ветки, в которой находятся вливаемые изменения. Оно, конечно, написано абзацем выше, но ведь в реальности мы работаем не по учебнику. Чтобы вспомнить требуемое имя, выполните команду

```
git branch
```

и теперь влейте изменения из ветки feat/entity-create в ветку main. Для этого убедимся, что мы находимся в ветке main (собственно, мы в этом только что убедились при помощи команды git branch – мы же видели звёздочку), и выполним команду

```
git merge feat/entity-create
```

Изменения влиты. Убедитесь в этом при помощи журнала изменений (с ключом oneline). Мы видим, что коммит с созданием сущности теперь присутствует и помечен как «(HEAD -> main, feat/entity-create)», т.е. принадлежит обеим веткам.

Обратите внимание: поскольку в ветку main не добавлялось новых коммитов с момента создания нашей ветки, Git выполнит оптимизированное слияние, называемое fast-forward (быстрое перематывание). Он просто «передвинет» указатель ветки main на наш последний коммит, не создавая дополнительного коммита слияния. История останется линейной и чистой.

Ветку feat/entity-create теперь можно удалить. Сделайте это:

```
git branch -d feat/entity-create
```

Ключ -d безопасен: Git не даст удалить ветку, если её изменения ещё не были слиты, защищая вас от потери данных.

Удаление слитых веток – это хорошая гигиена репозитория, которая не позволяет списку веток превратиться в бесконечную свалку старых названий.

На примере добавления сущности мы увидели, в общем-то, идеальное течение базового цикла работы с Git. Закрепим материал на примере удаления сущности.

Создайте ветку feat/entity-delete и перейдите на неё. Добавьте в конец файла документации строку

```
Удаление сущности – entity.delete
```

Сохраните изменения, проверьте состояние репозитория, посмотрите сделанные изменения, добавьте их к индексу и выполните коммит с сообщением «feat: удаление сущности».

Переключитесь на основную ветку, убедитесь, что в ней не видны изменения, внесённые в другой ветке. Посмотрите список веток, влейте ветку с удалением сущности в основную ветку и удалите её.

§ 3.4. Добавление из основной ветки в текущую

В ходе данной лабораторной работы мы уже вливали изменения из временной ветки в основную. Но иногда бывает нужно изменения, влитые в основную ветку, добавить в текущую рабочую ветку. Потренируемся в отработке такой ситуации.

Создадим ветку для документирования функций работы со свойствами сущностей – назовём её feat/properties – и перейдём в неё. Пусть это будет у нас «долгоиграющая ветка», в которой мы будем делать множество связанных друг с другом по смыслу изменений – это позволит нам отработать целый ряд связанных с этим ситуаций.

Добавьте в конец файла документации строки

```
# Свойство

Добавление свойства к сущности – entity.properties.add(property)
```

Сохраните изменения. Просмотрите их, добавьте к индексу, закоммитьте с сообщением «feat: добавление свойства».

Теперь, допустим, пока один сотрудник работал над добавлением свойства, другому поступило задание на исправление ошибки в одной из уже написанных функций (допустим, оказалось, что в функцию создания сущности нужно передавать список её свойств). Для исправления ошибки нам понадобится новая ветка, создаваемая также из основной – ведь создаваемая ветка на момент создания является точным клоном той ветки, из которой она создаётся, а работа с текущей веткой у нас ещё не завершена, нам не надо размазывать незавершённую работу по нескольким веткам. Так что ветку для исправления ошибки нужно создать из основной ветки, содержащей стабильное, согласованное, законченное (до некоторой «точки») состояние проекта.

Переключитесь в основную ветку, создайте, находясь в ней, ветку с названием «fix/entity-create-add-properties» и переключитесь в эту новую ветку. Найдите в файле с документацией строку, содержащую функцию entity.create, и измените описание функции с «entity.create» на «entity.create(properties)».

Сохраните изменения, просмотрите их и состояние репозитория, добавьте изменения к индексу и закоммитьте их с сообщением «fix: сущность создаётся со списком свойств».

Вернитесь в основную ветку и посмотрите список веток. Обратите внимание, что мы видим все три ветки – основную (помеченную как текущая), ветку с исправлением и ветку для документирования функций работы со свойствами. Влейте в основную ветку исправление функции создания сущностей и посмотрите журнал изменений (в режиме по одной строке на коммит). Мы видим, что последний коммит принадлежит веткам «main» и «fix/...», но не ветке «feat/...». Удалите ветку, которая была создана для только что влитого в основную ветку исправления.

Вернёмся в ветку, в которой мы трудимся над функциями работы со свойствами (если название не помните, используйте команду списка веток). Посмотрите журнал изменений. Мы видим, что изменение, содержащее исправление ошибки, в этой ветке не появилось. Как бы нам его добавить?

Если мы просто вольём основную ветку в текущую, у них появится общий коммит, в котором, в том числе, будут содержаться коммиты текущей ветки, то есть, незавершённая работа (каждый из этих коммитов, как предполагается, содержит некий завершённый кусок работы, но цельная фича ещё не готова к добавлению в основную ветку). Нам надо сделать так, как будто бы наша текущая ветка с новой фичей была создана не из того коммита основной ветки, из которого она по факту была создана, а из того, который уже содержит исправление ошибки. То есть, надо «пересадить» текущую ветку на последний коммит основной ветки. Иначе говоря – перебазировать её.

Мы уже знаем такую команду. Выполним её:

```
git rebase main
```

Текущая ветка перебазируется на последний коммит основной ветки.

Мы могли бы просто выполнить git merge main, но это создало бы в нашей ветке лишний «коммит слияния», который загрязняет историю. Команда git rebase работает элегантнее: она берет наши изменения и как бы «проигрывает» их заново поверх последнего коммита main. В итоге история нашей ветки остаётся идеально линейной и чистой, как будто мы начали свою работу уже после того, как хотфикс был внесен.

Посмотрите журнал изменений и сравните с его прошлым видом:

```
git log --oneline --graph
```

Вы видите, что теперь коммит с добавлением свойства идёт как бы после коммита с исправлением создания сущности, хотя хронологически он был создан ранее. Мы успешно пересадили ветку на новое место.

Обратите внимание: если бы мы в рамках данной лабораторной работы взаимодействовали с облачным репозиторием, перебазированные коммиты перестали бы соответствовать своим копиям в облаке, и команда git push без дополнительных флагов стала бы вызывать ошибку. Ситуацию облегчает использование флага force-with-lease. Эта команда скажет облаку: «Моя история изменилась, замени старую версию моей ветки на новую, но только если никто другой не успел запушить туда что-то новое».

§ 3.5. Разрешение конфликтов

До сих пор случалось так, что история одной из веток не противоречит истории другой – не возникает конфликтов. Однако на практике конфликты веток – увы, дело житейское. Потренируемся их исправлять.

Сейчас мы всё ещё находимся в ветке «feat/properties». Мы решили в этой ветке трудиться над всем множеством функций, посвящённых работе со свойствами. Добавим в конец файла с документацией строку

```
Удаление свойства – entity.properties.delete
```

Посмотрите состояние репозитория, сделанные изменения, добавьте изменения в индекс и закоммитьте из с сообщением «feat: удаление свойства».

Теперь, допустим, другой сотрудник начал работать над другой фичей. Переключитесь на основную ветку, создайте ветку «feat/object-add» и переключитесь на неё. Добавьте в конец файла с документацией строки:

```
# Объект

Добавление объекта – realm.add(entity)
```

Как обычно, посмотрите всё, что мы смотрим, добавьте изменения к индексу и закоммитьте с сообщением «feat: добавление объекта».

Сотрудник завершил работу. Переключитесь на основную ветку и влейте изменения из ветки «feat/object-add» (удалять её пока не будем, мы с ней ещё попрактикуемся).

Теперь, допустим, тот сотрудник, который работал над функциями свойств, получил информацию, что ему можно перебазировать свою ветку на свежие изменения в основной ветке. Попробуйте сделать это.

Что мы увидели? Мы увидели, что перебазирование не удалось из-за конфликта. В самом деле, что видит Git при попытке перебазирования? Он видит раздел про игровой мир, видит раздел про сущности, а дальше... А дальше в той ветке, которую мы перебазируем, идёт раздел про свойства, а в той, на которую перебазируем – про объекты. И как быть? Какой раздел должен идти сначала, какой потом? Или, быть может, один из них нужен, а другой не нужен? Ответ на эти вопросы лежит вне компетенции Git – к счастью (потому что представьте себе, что начнётся, если Git за нас будет принимать такие решения – как минимум возможна ситуация, когда он решит за нас не так, как нам бы хотелось). Поэтому тут мы должны вмешаться – то есть, разрешить конфликт вручную.

Git пишет в своём сообщении, что конфликт произошёл при обработке изменений в файле editor_api.md. Откроем его в редакторе. Мы увидим, что до конфликтного места содержимое общее, а дальше идёт строка <<<<< HEAD, потом та версия этого куска содержимого, на которую мы перебазируемся, далее строка =======, затем та версия, которую мы перебазируем, и, наконец, строка >>>>> и

сообщение того коммита, на перебазировании которого возник конфликт (в данном случае «feat: добавление свойства»).

Исправьте файл вручную – сделайте так, чтобы после корректной части содержимого сначала шёл раздел про объекты, потом про свойства, а служебные маркеры конфликтных частей удалите. После этого сохраните изменения.

Проверьте состояние репозитория. Мы видим, что перебазирование не завершено. Мы исправили конфликт, но надо запустить дальнейший ход перебазирования – дать Git понять, что мы удовлетворены изменениями и пока больше их не планируем, готовы продолжить перебазирование.

Для этого мы добавляем внесённые в процессе разрешения конфликта изменения к индексу при помощи git add. После этого снова проверим состояние репозитория. Git проверяет, не осталось ли ещё конфликтов. Поскольку их не осталось, он пишет, помимо прочего:

```
all conflicts fixed: run "git rebase --continue"
```

То есть, Git сообщает, что больше конфликтов пока не обнаружил, и говорит, какой командой мы можем продолжить процесс перебазирования. Выполним эту команду:

```
git rebase --continue
```

После этого Git, в зависимости от его версии и настроек, может предложить ввести сообщение для коммита, содержащего разрешение конфликта. Если он предложил это сделать, безо всяких изменений сохраним предложенный файл (это временный файл, нужный только для того, чтобы Git открыл редактор, передал туда то, что хочет, и принял от нас наши пожелания).

Git выводит сообщение о том, что перебазирование завершено успешно. Посмотрим журнал изменений. Мы видим, что наша ветка успешно перебазирована на коммит с добавлением объекта, ни одного коммита не пропало.

Убедимся, что возникший конфликт после его разрешения дальнейших последствий не имеет – добавим ещё какую-нибудь функцию.

Мы сейчас находимся в ветке «feat/properties».

Откроем файл editor_api.md. Кстати, заодно убедимся, что в нём присутствуют все изменения и из ветки main, и из ветки feat/properties. В раздел «Свойство» добавим строку:

```
Переименование свойства – entity.properties.rename(old\_name, new\_name)
```

В формате Markdown знак подчёркивания имеет служебное значение, поэтому мы это значение явным образом отменяем при помощи символа \.

Добавьте изменения к индексу и закоммитьте их с сообщением «feat: переименование свойства». Всё получилось.

§ 3.6. Тайник

Помните, мы не стали удалять ветку feat/object-add? Вот сейчас она нам пригодится для практики в работе с тайником.

Допустим, мы в какой-то момент допустили оплошность и начали вносить изменения не в той ветке, в какой бы хотели.

Переключитесь на ветку feat/object-add. Откройте файл editor_api.md и добавьте в раздел «Объект» строку

```
Изменение объекта – object.update
```

Сохраните файл.

И вдруг мы понимаем, что это изменение мы должны были делать не здесь. Ведь ветка-то называется feat/object-add, а у нас это уже не add, это уже update.

Как же быть? Удалять свою работу, создавать новую ветку, и писать эту строку документации заново?

По счастью, удалять свою работу и писать её заново не надо. Мы воспользуемся такой хитрой возможностью Git, как stash – тайник.

Изменения, которые мы уже сделали, но ещё не добавили в индекс, мы можем спрятать в тайник командой

```
git stash
```

После этого состояние папок и файлов вернётся к состоянию на момент последнего коммита – но сделанная работа не пропадёт, а просто переместится в тайник. Если мы сейчас проверим состояние репозитория или откроем файл в редакторе, мы последнего изменения не увидим, но в тайнике оно сохранилось.

Проверим список изменений, которые находятся в тайнике, при помощи команды

```
git stash list
```

Мы видим, что в тайнике находится одно изменение.

А можно ли его вернуть из тайника? Конечно. Выполним команду

```
git stash pop
```

Мы снова увидим в состоянии репозитория наличие изменений, а в редакторе мы увидим сделанные нами правки.

Тайник устроен как своего рода стек – командой git stash мы можем складывать туда изменения, а команда git stash pop каждый раз извлекает из тайника последнее изменение и применяет его к текущему состоянию файлов и папок. Если мы ещё

раз посмотрим на список изменений, хранящихся в тайнике, мы увидим, что он опустел.

Снова положите эти изменения в тайник и посмотрите на список того, что там лежит – мы снова увидим в тайнике одно изменение.

А можно ли получить какую-то информацию о последнем изменении, записанном в тайник? Да, для этого служит команда

```
git stash show
```

Выполнив её, мы видим, что это изменение затрагивает только файл editor_api.md и состоит в добавлении одной строки. Но можно получить и всё содержимое изменения – для этого нужно использовать ключ p (от слова patch):

```
git stash show –p
```

Мы видим добавленную строку и её ближайшее окружение.

Обычно в стеке можно не только вынуть последний объект, но и просто его посмотреть. Так же и с тайником – можно применить последнее хранящееся в нём изменение, не убирая его из тайника, при помощи команды

```
git stash apply
```

Выполните её. Посмотрев состояние репозитория и список содержимого тайника, мы увидим, что изменение применилось, но из тайника не исчезло.

А если его ещё раз положить в тайник?

А попробуйте. Мы увидим, что теперь оно хранится в тайнике дважды: один раз под именем stash@{0}, а другой раз под именем stash@{1}. Мы можем посмотреть краткую или полную информацию о любом из них, указав после слова show имя изменения.

Тайник работает, напомним, как стек – а пользуясь бытовыми аналогиями, как стопка тарелок: команда stash кладёт тарелку наверх, а pop снимает самую верхнюю. Поэтому stash@{0} – это всегда самое последнее (верхнее) сохранение.

Ладно, но нам не нужно два одинаковых изменения в тайнике. Мы можем как-то удалить одно из них, не применяя его?

Да, можем. Любое изменение из тайника можно удалить при помощи команды

```
git stash drop имя_изменения
```

Для последнего изменения имя можно не указывать.

Оставьте в тайнике только одно изменение и убедитесь в том, что сделали всё правильно.

Итак, теперь у нас в ветке feat/object-add и в рабочем состоянии репозитория изменений нет, а то изменение, которое мы начали делать не в той ветке, теперь хранится в тайнике. Теперь мы можем это изменение отсадить в нужную ветку.

Перейдите в основную ветку, а ветку feat/object-add удалите – она нам больше не нужна. Создайте ветку для работы с остальными функциями объектов – назовите её feat/objects-etc. Перейдите в неё.

Посмотрите список содержимого тайника. Мы видим, что сделанное ранее в той ветке, которой уже нет, изменение так там пока и лежит. Извлеките его из тайника. Теперь оно применилось в уже новой ветке, а из тайника убралось.

Мы перенесли изменение из одной ветки в другую, не переписывая его заново.

Обратите внимание, что это получилось в том числе потому что мы не успели скоммитить это изменение (и даже добавить его к индексу).

Добавьте сделанное изменение к индексу и закоммитьте его с сообщением «feat: изменение объекта».

Пока мы закончили с объектами. Перейдите в основную ветку и влейте в неё ветку, с которой мы сейчас работали. Не удаляйте её, мы ещё будем добавлять туда описания новых функций.

Перейдите в ветку, где мы документируем функции работы со свойствами, и перебазируйте её на свежее состояние основной ветки. Опять получится конфликт (по той же причине) – разрешите его. Обратите внимание: при разрешении конфликта во время rebase Git останавливает процесс на **первом** коммите вашей ветки, который вызывает конфликт. После того как вы его разрешите (через git add и git rebase --continue), Git автоматически попытается наложить следующие коммиты вашей ветки. Если они не конфликтуют с новыми изменениями в main, они применятся тихо и без дополнительных вопросов.

§ 3.7. Параллельная работа в двух ветках

Потренируемся в обработке ситуаций, когда работа над одним и тем же файлом ведётся двумя разными сотрудниками в двух разных ветках.

Сейчас мы в ветке feat/properties. Добавим описание функции, которая будет указывать, какое свойство сущности будет её идентификатором: в раздел «Свойство» добавьте строку

```
Задание идентификатора – entity.set\_id(property)
```

(не забываем про экранирование подчёркивания).

Добавьте изменение к индексу и закоммитьте его с сообщением «feat: задание идентификатора».

Добавьте следом описание функции, которая будет создавать свойство сущности, указывающее на объект другой сущности (например, у сущности «предмет» может быть свойство «рюкзак», которое указывает на сущность «рюкзак»):

```
Ссылка на другую сущность – entity.belongs\_to(property, entity)
```

Аналогично, закоммитьте как «feat: задание ссылки».

Теперь мы другой сотрудник, который в это же время работает с другой веткой. Перейдём в ветку feat/objects-etc. В раздел «Объект» добавим строку

```
Удаление объекта – object.delete
```

Закоммитьте изменения с сообщением «feat: удаление объекта».

А теперь мы – руководитель проекта, который должен слить изменения от обоих сотрудников. Перейдём в основную ветку.

Влейте в основную ветку изменения из ветки про объекты. Если всё было сделано правильно, слияние произойдёт без проблем. После этого удалите ветку про объекты, она больше не понадобится.

В этот момент, допустим, внезапно оказалось, что нам нужны возможности по переименованию не только свойств, но также сущностей и собственно игрового мира. Для исправления ошибок создадим из основной ветки ветку fix/add-missing-renames и перейдём в неё.

В раздел «Игровой мир» добавьте строку

```
Переименование мира – realm.rename
```

Закоммитьте с сообщением «fix: переименование мира».

За это время сотрудник, работавший со свойствами, наконец, закончил свою работу. Перейдите в ветку feat/properties. Прежде, чем вносить последнюю правку, перебазируйте свою ветку на свежий коммит основной ветки (разрулив конфликт, если таковой возник).

Последняя функция работы со свойствами будет устанавливать на свойство сущности правило, чтобы значение этого свойства у всех объектов было разным.

Добавьте в раздел «Свойство» строку

```
Уникальность – entity.unique(property)
```

Закоммитьте изменение с сообщением «feat: уникальное свойство».

§ 3.8. Редактирование истории изменений

Помимо прочего, Git позволяет нам редактировать историю изменений.

ВНИМАНИЕ!!! Эти функции следует использовать ТОЛЬКО в тех случаях, когда либо затрагиваемая часть истории не отправлялась в облако, либо она относится только к ветке, с которой не работали и не планируют работать другие сотрудники!

Команда git reset позволяет перемещаться по истории изменений. При помощи команды

```
git reset хэш
```

мы можем переместиться на состояние, сохранённое в коммите с заданным хэш-кодом (хэш можно указать краткий).

Можно также переместиться на n коммитов от последнего. Для этого служит команда git reset HEAD~n, где n – число, на сколько коммитов нужно переместиться.

Переместитесь на предпоследний коммит текущей ветки:

```
git reset HEAD~1
```

Те изменения, которые были в последнем коммите, теперь становятся как бы не помещёнными в индекс. Если мы посмотрим журнал изменений, мы не увидим в нём последнего коммита (про уникальное свойство), а если посмотрим состояние репозитория, мы увидим, что соответствующее изменение стало не добавленным в индекс. Если мы посмотрим журнал изменений, мы не увидим в нём последнего коммита, а git status покажет файлы с изменениями красным цветом как изменённые, но не готовые к коммиту.

У этой команды есть ещё два варианта. Данный вариант как бы «средний», есть более «жёсткий» и более «мягкий». Эта же команда с ключом --hard выполняет «жёсткий» вариант – изменения не попадают в рабочее состояние файлов и папок, а вообще пропадают. А эта же команда с ключом --soft выполняет «мягкий вариант» – изменения остаются добавленными в индекс.

После того, как мы выполнили git reset, мы можем снова добавить изменения в индекс и снова их закоммитить. Сделайте это, используя то же сообщение коммита.

Способ перехода на предыдущий коммит через git reset фактически удаляет коммит (или несколько) из истории. Есть способ вернуться к предыдущему состоянию, не удаляя последний коммит, а создавая коммит с противоположным изменением – это более безопасно, т.к. всё, что было сделано, в истории остаётся. Для этого служит команда git revert.

Чтобы таким образом откатить последнее изменение, выполните команду

```
git revert HEAD
```

В истории создастся коммит с изменениями, противоположными тем, которые были в последнем коммите. При этом сообщение коммита мы можем отредактировать.

Используя команды git show и git show HEAD~1, убедитесь, что изменения, выполненные последним (теперь) и предпоследним (бывшим последним) коммитами противоположны друг другу.

Итак, два способа «провернуть фарш назад» – это git reset и git revert. Однако, есть ещё и третий – интерактивное перебазирование. Эта процедура позволяет взять несколько последних коммитов и выполнить с ними те или иные действия, такие, как удаление коммита, изменение его сообщения, склеивание нескольких коммитов в один и т.д. Например, для того, чтобы выполнить интерактивное перебазирование n последних коммитов, нужно выполнить команду git rebase --interactive HEAD~n.

Начните интерактивное перебазирование двух последних коммитов – выполните для этого команду

```
git rebase --interactive HEAD~2
```

Когда вы выполните эту команду, Git откроет настроенный редактор (Vim, Nano или другой), в котором будут перечислены хэши и сообщения последних нескольких коммитов (в данном случае двух), а перед каждым из них будет стоять та или иная предлагаемая команда – по умолчанию pick (в данном случае – оставить коммит без изменений). Под перечнем коммитов вы видите шпаргалку по доступным в этом режиме командам (всего их примерно дюжина). Сами коммиты перечислены в порядке от самого старого к самому новому (т.е. в порядке, обратном тому, как показывает их команда git log).

Обоим перечисленным коммитам замените команду pick на команду d (или drop) – удаление коммита. Для этого в строках, соответствующих каждому из двоих коммитов, замените начальное pick на d или drop, после чего сохраните файл и выйдите из редактора.

После того, как вы вышли из редактора, Git поочерёдно отрабатывает команды, указанные для каждого из коммитов. Убедитесь в том, что оба коммита удалены, при помощи журнала изменений.

А можно ли выполнить интерактивное перебазирование всей истории, с самого первого коммита? Можно. Для этого вместо указания стартового коммита используется ключ root. Начните интерактивное перебазирование всей истории, выполнив команду

```
git rebase --interactive --root
```

Мы видим, что Git вложил в редактор всю историю, начиная не с момента создания ветки, а с момента создания самого первого коммита репозитория.

ВАЖНО: Никогда не используйте --root или не меняйте коммиты, которые уже находятся в ветке main или были отправлены в облако (origin/main). Это переписывает общую историю и сломает репозитории всех ваших коллег. Интерактивный ребейз – это инструмент для «причесывания» вашей личной, локальной ветки перед тем, как вы покажете её другим.

Давайте выполним замену сообщения у коммита, с которого началась ветка работы со свойствами – сейчас у него сообщение «feat: добавление свойства». Команда замены сообщения – это команда r, или reword. Найдите строку с этим коммитом и замените в ней команду pick на r.

Сохраните текст в редакторе и выйдите из него. Git начнёт выполнять команды одну за другой. Пока следуют команды pick, он будет молча использовать соответствующие коммиты как есть, а дойдя до команды r при соответствующем коммите, опять откроет редактор, в котором будет текущее сообщение данного коммита.

Замените сообщение на «feat: добавление свойства к сущности», сохраните текст и выйдите из редактора. Git заменит сообщение коммита и продолжит выполнение команд, заданных при интерактивном перебазировании. По окончании процесса посмотрите журнал изменений – вы увидите, что у соответствующего коммита изменилось сообщение.

Однако может возникнуть проблема – если при интерактивном перебазировании вы изменили какой-либо из коммитов основной ветки, более ранних, чем начало вашей текущей ветки, то вы уже не сможете влить эту ветку в основную ветку – у них различается история. В некоторых старых версиях Git это происходило, к слову сказать, даже если мы не меняли коммиты основной ветки, просто вследствие самого факта, что произошло интерактивное перебазирование всей истории. Чтобы вернуть нашу ветку на место, перед выполнением слияния нужно снова перебазировать её на основную ветку при помощи git rebase main.

Мы попробовали при помощи интерактивного перебазирования удалять коммиты и менять их сообщения. Теперь попробуем слить несколько коммитов в один.

Начните интерактивное перебазирование, начиная от того места, где появилась текущая ветка (в данном случае, если всё сделано правильно, это HEAD~5). Для всех коммитов, кроме самого первого (который «feat: добавление свойства к сущности»), замените команду pick на s или squash. Эта команда склеивает тот коммит, на котором она стоит, с предыдущим – получившийся объединённый коммит будет содержать изменения из обоих коммитов, а итоговое сообщение Git предложит создать, когда отработает склеивание (конечно, если склеивается сразу много коммитов в один, он запросит для них новое сообщение только один раз).

Сохраните изменения и выйдите из редактора. Git выполнит склеивание и предложит ввести новое сообщение для итогового коммита. По умолчанию в качестве такого сообщения он предложит текст, состоящий из всех сообщений

склеенных воедино коммитов. Согласитесь на это – просто сохраните текст, закройте редактор и дождитесь окончания интерактивного перебазирования.

Посмотрите журнал изменений. Вы увидите, что все коммиты данной ветки слились в один. В однострочном режиме вы увидите только первую строку итогового сообщения, в полном режиме увидите все строки. Посмотрите журнал изменений в полном режиме, но не целиком, а только одного (последнего) коммита – для этого используйте ключ -1:

```
git log –1
```

Вы увидите все строки сообщения.

Теперь мы готовы вливать ветку свойств в основную ветку. Перейдите в основную ветку и выполните слияние, после чего, наконец-то, удалите ветку свойств.

А в это время другой сотрудник продолжает работу по добавлению функций переименования. Перейдите в ветку fix/add-missing-renames.

Пока сотрудник отсутствовал, в основной ветке появился коммит – мы ведь выполнили слияние ветки про свойства. Перебазируйте текущую ветку на свежий коммит основной ветки. Посмотрите журнал изменений в режиме по одной строке на коммит, ограничившись пятью последними коммитами (флаг, соответственно, -5).

В раздел «Сущность» добавьте строку

```
Переименование сущности – entity.rename
```

Закоммитьте изменение с сообщением «fix: переименование сущности».

Ещё раз посмотрите последние пять изменений в однострочном режиме. Давайте объединим два коммита данной ветки в один. Начните интерактивное перебазирование двух последних коммитов (кстати, вместо --interactive можно использовать просто -i), склейте эти два коммита в один. Сохраните текст и закройте редактор. Git выполнит склейку и запросит новое сообщение, предложив как вариант комбинацию сообщений обоих склеиваемых коммитов. Не удаляя эти сообщения, добавьте сверху строку:

```
fix: добавлены недостающие переименования
```

Сохраните текст и выйдите из редактора. Git сделает остальное.

Посмотрите в журнале последний коммит. Всё хорошо. Вернитесь в основную ветку, слейте изменения из ветки, в которой только что были, и удалите её. Если всё было сделано правильно, осталась только основная ветка.

Посмотрите журнал изменений в однострочном режиме.

§ 3.9. Список игнорирования

В принципе, работа почти закончена. Давайте только попрактикуемся в составлении списка игнорирования.

Создайте ветку с названием feat/add-gitignore и перейдите в неё. При помощи команды touch создайте файл с секретным содержимым – например, как часто бывает в веб-проектах, пусть это будет файл с названием «.env». Добавьте в него секретную информацию – например, такую:

```
API_KEY=1846349876546
```

Это будет представлять собой, например, ключ к API некоего используемого нами сервиса.

Посмотрим состояние репозитория. Мы видим, что файл с секретной информацией в нём указан как изменение. Если бы мы, не подумав, неосторожно выполнили команду «git add .», мы бы тем самым открыли секрет всем, кто получит доступ к репозиторию. Чтобы этого не случилось, добавим файл в список игнорирования.

Создайте файл с названием «.gitignore» и откройте его в редакторе. Добавьте в него строку

```
.env
```

Сохраните файл .gitignore и закройте редактор.

Посмотрите ещё раз состояние репозитория. Теперь вы увидите в списке изменений новый файл .gitignore, но файл .env уже не увидите – Git читает содержимое файла .gitignore, видит .env в списке игнорирования и пропускает его. Теперь вы файл .env случайно не раскроете кому попало.

ВАЖНО: это работает только потому что .env ещё ни разу не добавлялся в индекс, ведь .gitignore работает только с новыми, неотслеженными файлами. Если вы по ошибке уже добавили секретный файл в индекс (git add), простое добавление его в .gitignore не поможет – Git продолжит следить за ним. В таком случае нужно сначала удалить файл из индекса, сохранив его на диске: git rm --cached .env, и только потом правило в .gitignore вступит в силу.

В реальных проектах в файле .gitignore могут быть десятки строк с самыми различными файлами и целыми их категориями.

Закоммитьте сделанные изменения (убедитесь перед коммитом, что в индексе только файл .gitignore) с сообщением «feat: список игнорирования».

Убедимся, что на обычные файлы с расширением .env это не повлияло. Добавьте файл с названием example.env. Убедитесь, что команда просмотра состояния репозитория видит это изменение. Добавьте в файл example.env строку

```
API_KEY=
```

Таким образом, если в вашей команде есть такое соглашение (хотя индустриальным стандартом de facto является вариант .env.example либо .env.template), вы даёте понять коллегам, что у себя они должны создать свой файл .env, в котором должно быть секретное значение API_KEY.

Добавьте изменение в индекс. Убедитесь, что example.env в индекс попал, а .env не попал.

Закоммитьте изменения с сообщением «feat: пример секрета» и ещё раз посмотрите пять последних коммитов в однострочном режиме. Склейте два последних коммита, итоговым сообщением сделайте «feat: добавлены список игнорирования и пример секрета».

Мы закончили со списком игнорирования. Перейдите в основную ветку, выполните слияние и удалите ставшую излишней ветку.

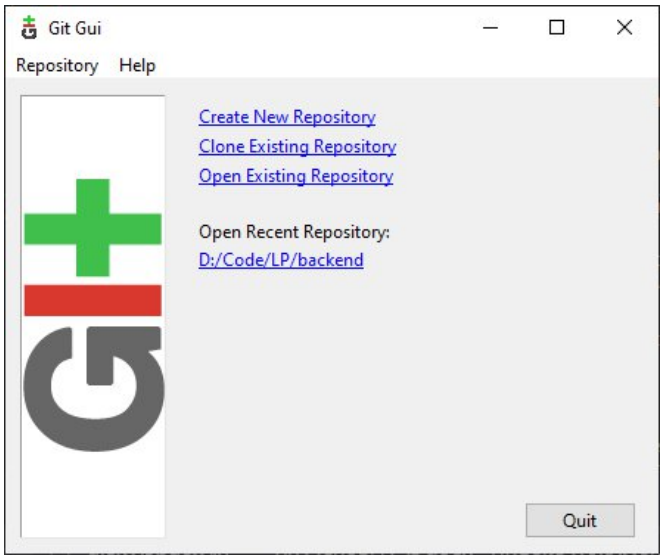
Посмотрите журнал изменений. Не правда ли, мы хорошо поработали? Теперь, когда вы уверенно владеете командами Git, пришла пора посмотреть, как те же самые операции выглядят в различных графических интерфейсах.

Часть 4. Графические клиенты Git

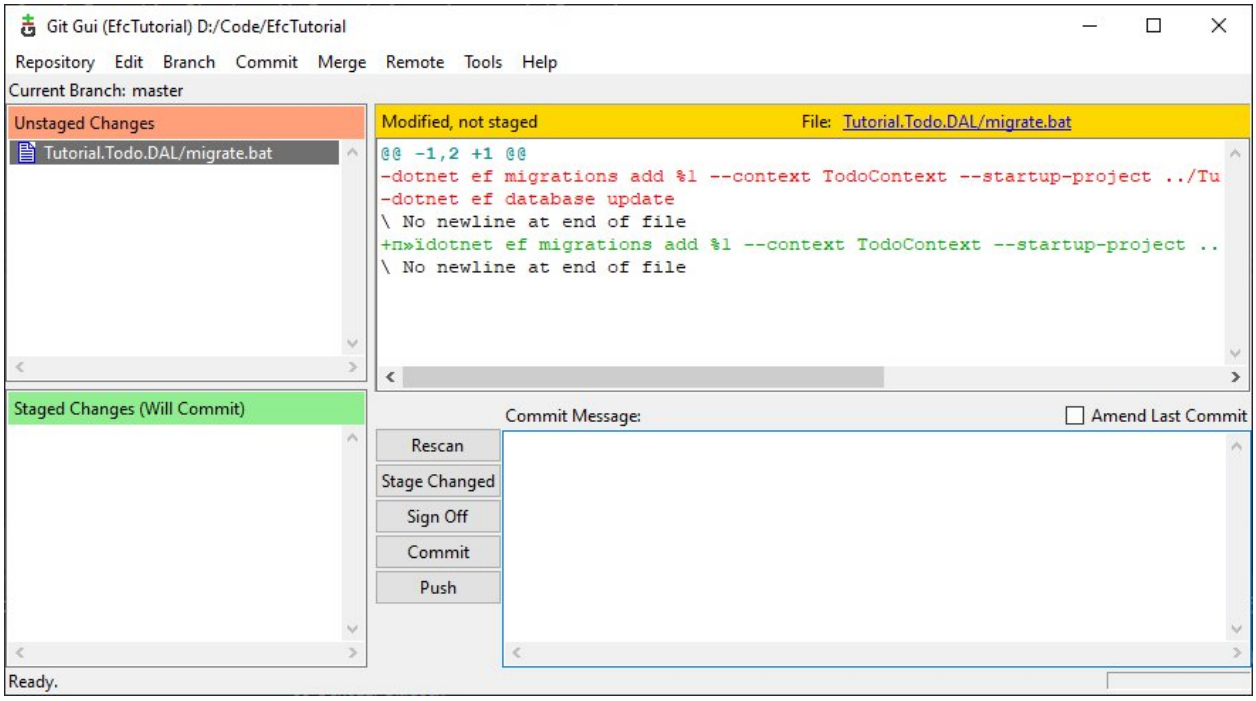
§ 4.1. Git GUI

До сих пор мы работали с Git в командной строке. Это полезно для того, чтобы понимать, какие вообще команды Git предоставляет, какие есть тонкости в их использовании и как можно автоматизировать работу с Git (скажем, при создании конвейеров непрерывной поставки кода). Однако на практике гораздо чаще основные, самые базовые функции Git используют при помощи тех или иных графических инструментов. Вполне естественно начать их перечисление с инструмента, входящего в комплект Git для Windows, а именно, с Git GUI (скорее всего, аббревиатура GUI вам знакома, но на всякий случай, GUI – это graphical user interface, графический интерфейс пользователя). Это не самый используемый из клиентов Git, предоставляющих графический интерфейс, но раз уж он входит в комплект, то мы с него и начнём. На самом деле, Git GUI есть и в macOS (доступен через Homebrew), и в Linux, но там он используется «из коробки» ещё реже, чем в Windows. Интерфейс пользователя этих клиентов под разные операционные системы по факту практически идентичен, поэтому далее мы будем описывать вариант только под Windows, но скриншоты будут понятны на любой из перечисленных систем.

Программа Git GUI представляет собой обычное оконное приложение с достаточно простым и непритязательным графическим интерфейсом пользователя. При запуске она прежде всего спрашивает, с каким репозиторием мы хотим работать – мы можем создать новый локальный репозиторий (фактически при этом запрашивается папка, а затем выполняется команда git init), можем склонировать существующий репозиторий (git clone) или перейти к уже существующему локально репозиторию. Также на стартовом экране отображается список недавно открывавшихся при помощи данной программы локальных репозиториев.



После выбора репозитория Git GUI переходит в режим отображения изменений. В верхней части окна слева отображается список файлов с изменениями, ещё не внесёнными в индекс, а справа показывается, какие именно изменения были сделаны в выбранном слева файле. В нижней части окна слева отображается список уже внесённых в индекс изменений, а справа расположено поле для ввода сообщения коммита. Также между двумя нижними зонами располагаются кнопки для обновления этой информации, внесения изменений в индекс, создания коммита и отправки его в облако:

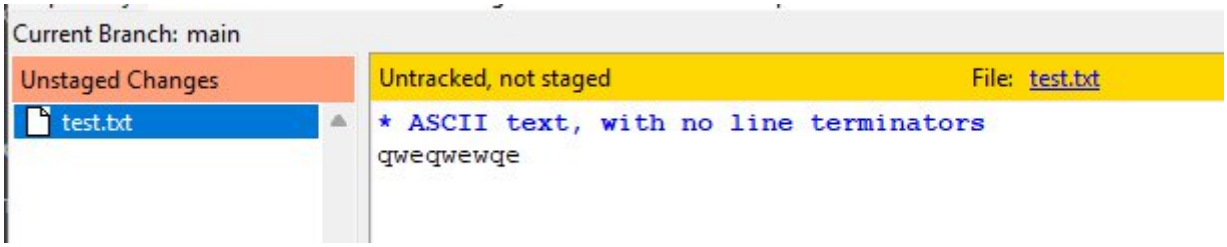


Прочие действия доступны через меню. Конечно, там представлен не весь спектр возможных действий, поэтому при необходимости к ним можно добавить желаемые команды при помощи меню «Tools» («Инструменты»). Например, можно сделать так, чтобы через меню Tools → Create Archive... можно было создать архив с кодом (аналог команды git archive), а через Tools → Git Bash мгновенно открыть терминал в папке текущего репозитория, если нужно выполнить сложную команду вручную.

Для закрепления материала попробуем кое-что на практике.

Перейдите в репозиторий, с которым мы работали в третьей части (про игровой мир) и создайте в нём новый файл, например, test.txt. Прямо в Блокноте или другом текстовом редакторе напишите в нём несколько слов и сохраните файл.

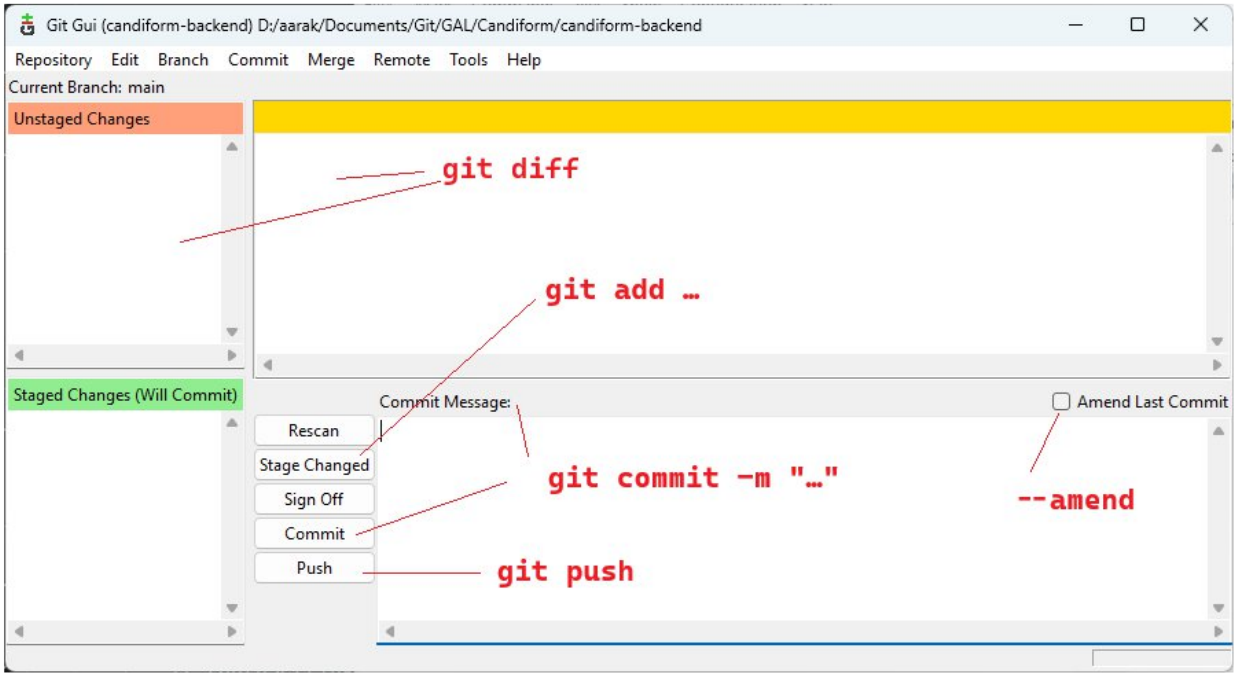
Теперь откройте репозиторий в Git GUI и найдите этот свежесозданный файл в панели Unstaged Changes. Нажмите на него – вы увидите сделанные в файле изменения (в данном случае всё содержимое) справа:



Нажмите кнопку Stage changed. Файл переедет в панель Staged Changes. В панель Commit message введите сообщение «test: эксперименты с Git GUI» и нажмите кнопку Commit.

Вы выполнили базовый цикл при помощи Git GUI.

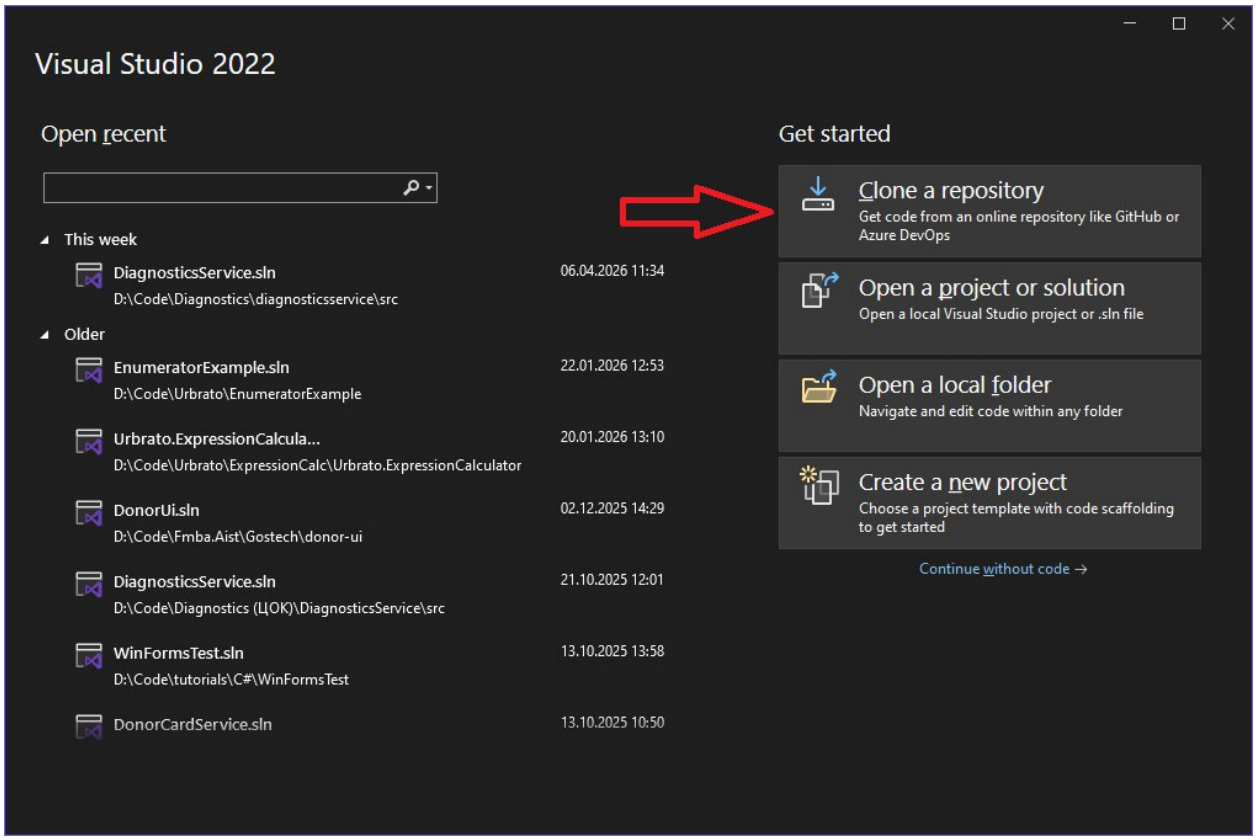
Напоследок наглядно сопоставим некоторые элементы интерфейса с уже известными вам командами:



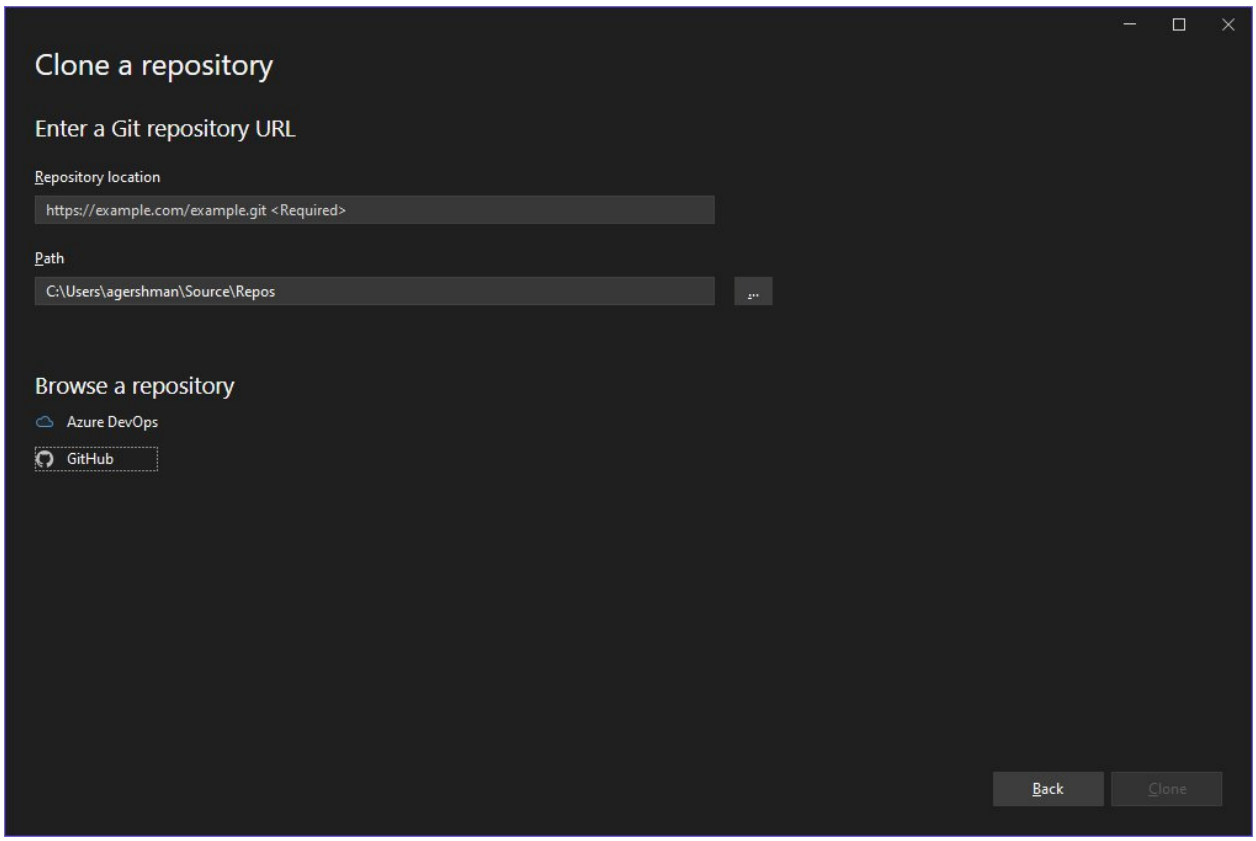
§ 4.2. Встроенные возможности Visual Studio

В интегрированной среде разработки Microsoft Visual Studio, предназначенной в основном (но не только) для разработки приложений на языке C#, уже давно существуют встроенные инструменты работы с системами контроля версий. Однако вплоть до 2013 года они были рассчитаны на централизованные системы с пессимистической блокировкой, что делало их малоприменимыми в сложившихся уже на тот момент реалиях. Ситуация улучшилась с выходом версии Microsoft Visual Studio 2013 Update 1. Ещё какое-то время ушло на исправления и доработки, и к версии Microsoft Visual Studio 2022 мы имеем уже вполне приемлемый продукт, хотя и не без проблем (в частности, как показывает практика, не всегда корректно отрабатывает разрешение конфликтов).

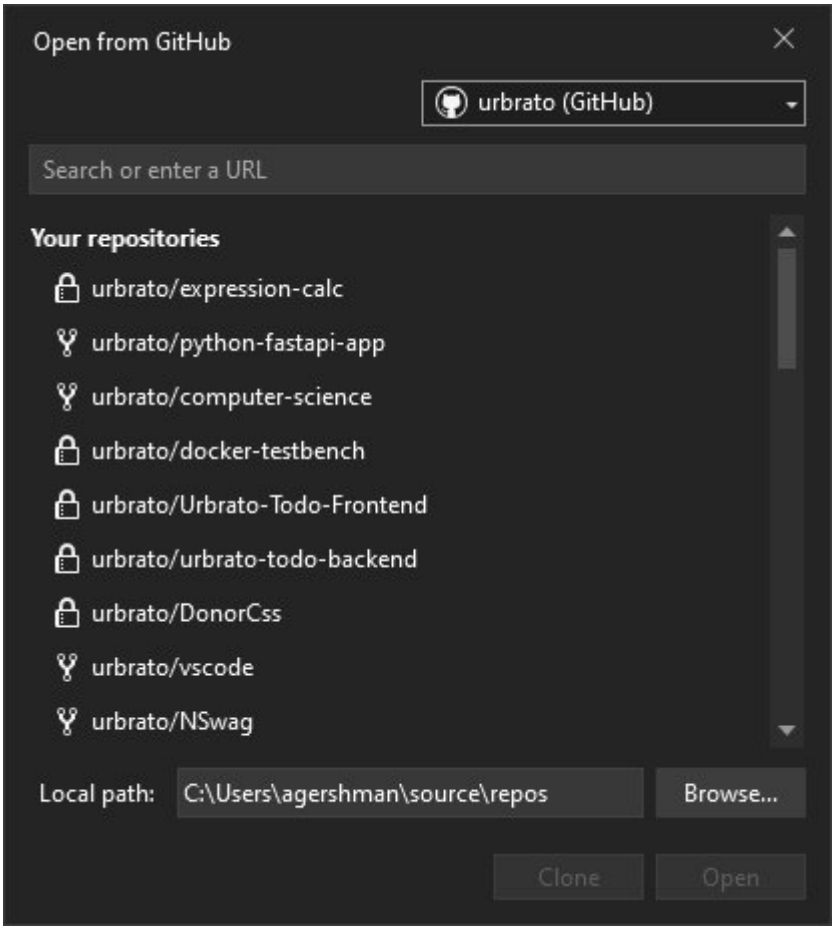
Когда мы открываем Microsoft Visual Studio, приложение сразу предлагает нам варианты, что можно сделать: создать новый проект, открыть существующий, продолжить работу с ранее открытым проектом и т.д. Одна из доступных возможностей – это клонирование репозитория из облачного сервиса.



Если ей воспользоваться, откроется окно параметров клонирования. В этом окне мы можем указать ссылку на облачный репозиторий (в том же формате, в котором мы её указываем для команды git clone) и выбрать папку, куда клонировать репозиторий локально:

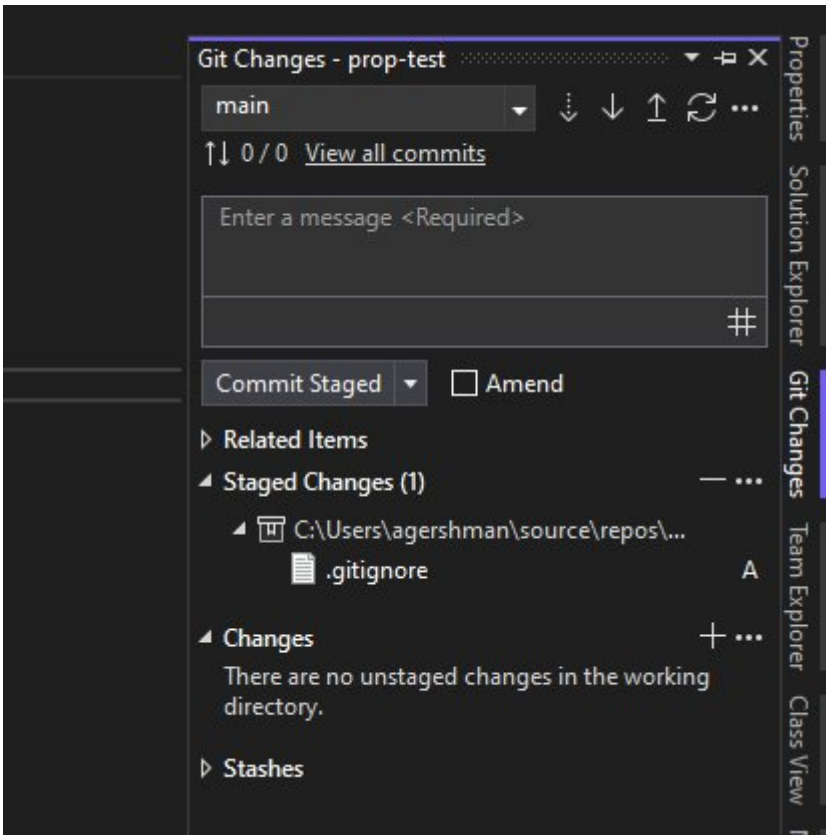


Кроме того, для некоторых облачных сервисов (на текущий момент к ним относятся только Azure DevOps и GitHub) предоставляется возможность вместо копирования ссылки выбрать репозиторий из списка:



После этого производится клонирование выбранного репозитория, и пользователь может работать с рабочей версией кода.

Среди боковых панелей, помимо известных с самых первых версий панелей свойств, обзора решения и прочих, появляется также панель работы с Git:

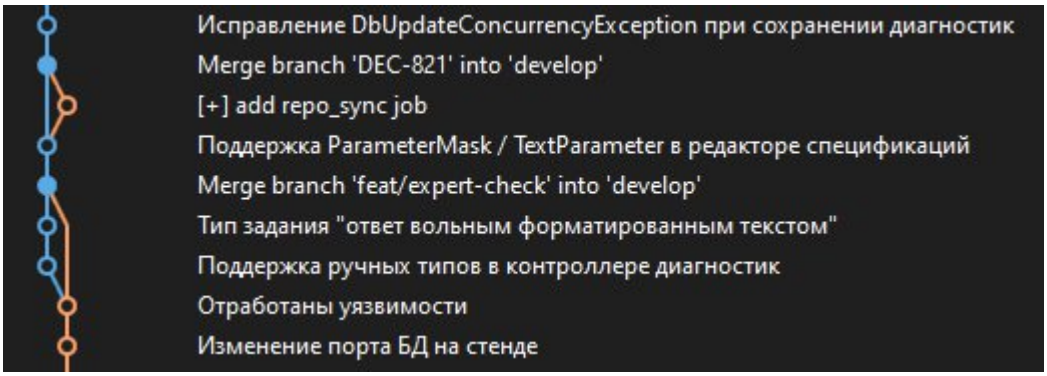


В панели Git Changes (Изменения Git) изменения отображаются в двух зонах: Changes (все изменения, ещё не в индексе) и Staged Changes (готовые к коммиту). По умолчанию новые или изменённые файлы попадают в Changes. Чтобы добавить файл в индекс, нажмите значок + рядом с ним (или щёлкните правой кнопкой → Stage). Чтобы убрать из индекса, нажмите – (или Unstage). Это прямой аналог команд git add и git reset, которые мы разбирали ранее. Однако в настройках может быть включён т.н. автостейджинг – когда по умолчанию все изменения попадают сразу в индекс, и мы можем убрать их оттуда вручную. Кстати, если мы видим новый файл, который, по-хорошему, нужно добавлять не в индекс, а в список игнорирования, мы можем щёлкнуть на нём правой кнопкой и выбрать пункт «Add to .gitignore». Visual Studio внесёт новое правило в список игнорирования и применит его.

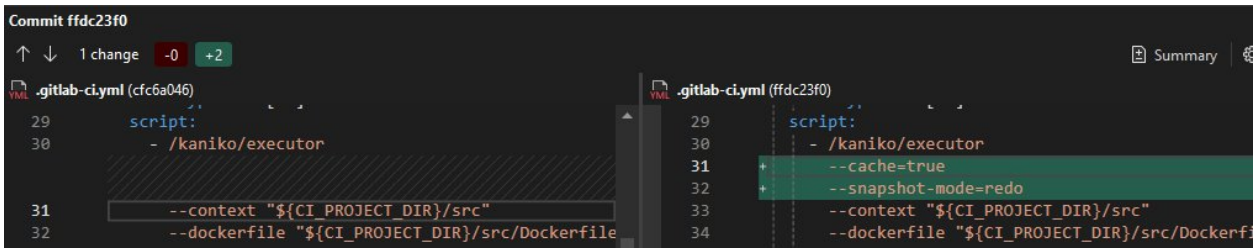
В этой же панели Git Changes мы можем просматривать содержимое тайника (Stashes), ввести сообщение для коммита и скоммитить изменения. Вверху мы видим выпадающий список веток и кнопки работы с удалённым репозиторием – это кнопки для команд fetch, pull и push, а также кнопка sync, которая выполняет pull, а затем, если pull прошёл успешно, выполняет push. Это удобно для тех случаев, когда вы уверены, что во время пулла не возникнет конфликта (например, работаете в своей ветке, как предполагает хороший стиль). Однако если есть

вероятность, что в удалённом репозитории есть чужие изменения, вытягивание которых вызовет конфликт в локальном репозитории, лучше выполняйте pull и push по отдельности, чтобы контролировать каждый шаг.

Тут же мы можем увидеть список уже выполненных коммитов вместе с графическим отображением дерева веток:

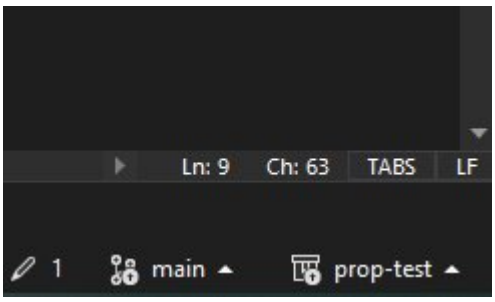


Двойной щелчок на любом из коммитов вызовет вкладку сравнения данного коммита с предыдущим, что позволяет быстро увидеть, что было сделано в этом коммите:



В этом окне мы можем не только постепенно прокручивать код по вертикали, но также можем и быстро перескакивать от одного изменения к другому, если их больше одного.

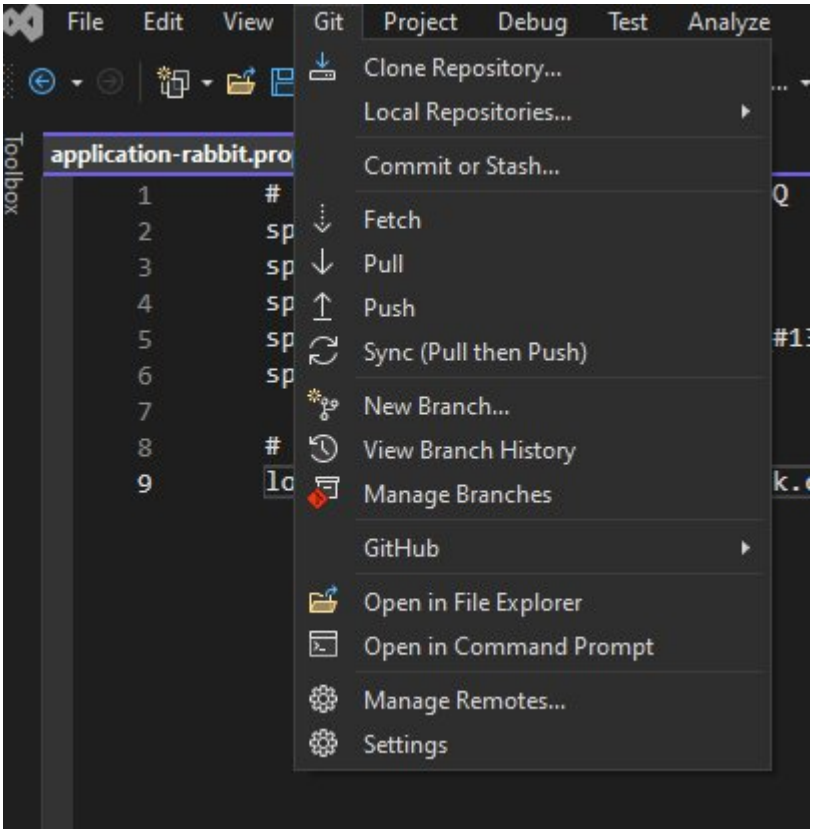
В правом нижнем углу основного окна мы также видим выпадающие списки для выбора текущей ветки и текущего репозитория:



На самом деле, при выборе выпадающего списка веток открывается не просто список, а панель, в которой мы также можем и создать новую ветку.

Если нам не хватает этих визуальных возможностей, на этот случай появилось целое меню Git (по умолчанию расположенное между View и Project), в котором

сгруппированы поддерживаемые MS Visual Studio визуальные команды для работы с Git:



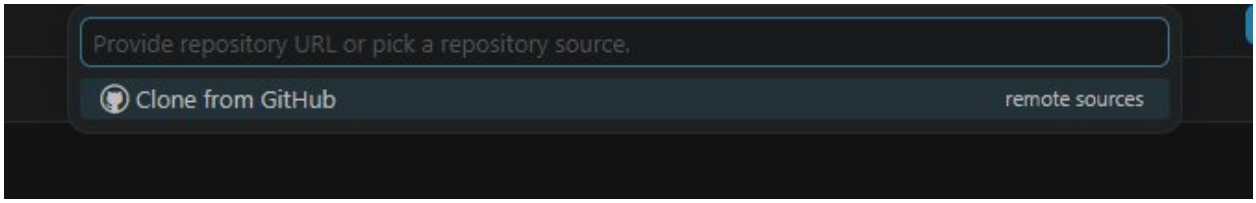
Как мы видим, помимо прочего есть также возможность открыть командную строку и работать с Git из неё, если визуальных возможностей нам не хватает.

§ 4.3. Встроенные возможности Visual Studio Code

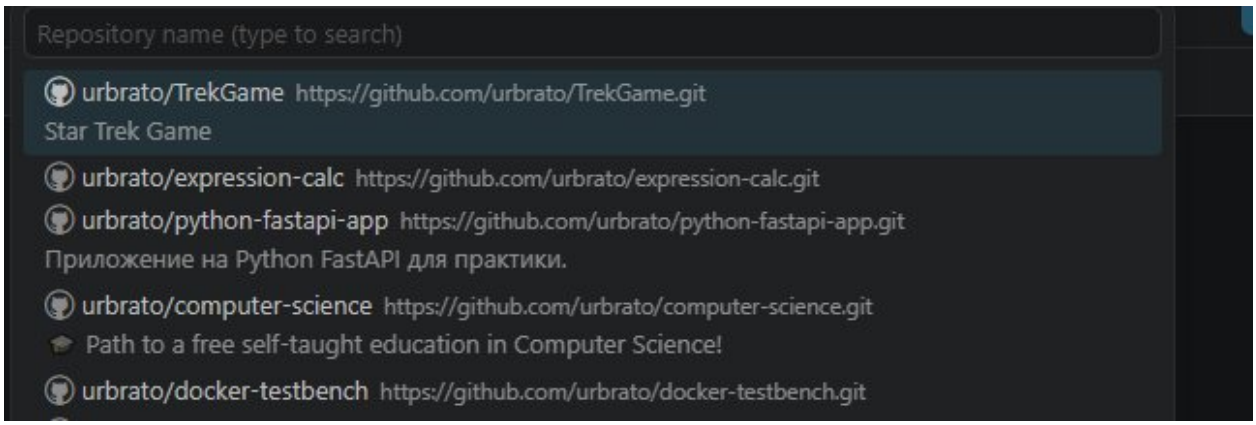
В отличие от Microsoft Visual Studio, являющейся полновесной средой разработки (преимущественно на C#), Microsoft Visual Studio Code позиционируется как текстовый редактор, превращаемый в среду разработки на том языке, который желателен, при помощи устанавливаемых расширений. Однако работа с Git в Visual Studio Code реализована «из коробки», хотя есть расширения, делающие эту работу комфортнее.

Так же, как Visual Studio, при открытии нового окна Visual Studio Code предлагает, помимо прочих возможностей, клонирование репозитория:

Однако дальше человек, привыкший к интерфейсу Visual Studio, может на некоторое время «зависнуть» – обычно люди ожидают какого-то сколько-нибудь заметного диалогового окна (например, в центре экрана). Но Visual Studio Code тут обманывает ожидания и вместо большого диалога показывает едва заметную панельку в верхней части своего окна:

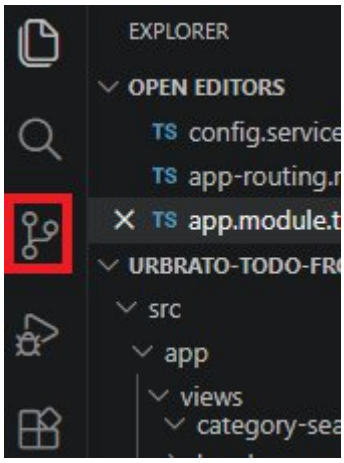


Так же, как и в случае Visual Studio, мы можем вставить (или вбить руками) url, предлагаемый облачным сервисом, а можем воспользоваться средствами интеграции с GitHub, если мы клонируем именно с этого сервиса. Если выбрать клонирование с GitHub, можно не копировать url вручную, а просто выбрать нужный репозиторий из списка:



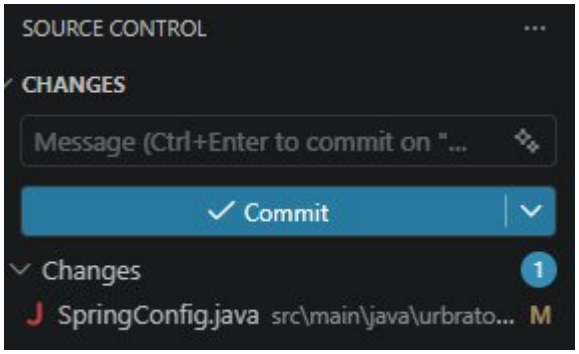
Запрос ввода пароля к приватному ключу SSH также появляется в верхней части окна в такой же незаметной панельке, это стандартный интерфейс пользователя Visual Studio Code. Если этого не знать, пользователь может некоторое время потратить на бесполезное ожидание стандартного модального окна «как у всех».

В Visual Studio Code боковые панели переключаются при помощи ряда кнопок в левой части окна. На каждой кнопке нанесён мнемонический значок. Значок кнопки, открывающей панель Source Control, через которую происходит работа с Git, намекает на работу с ветками:



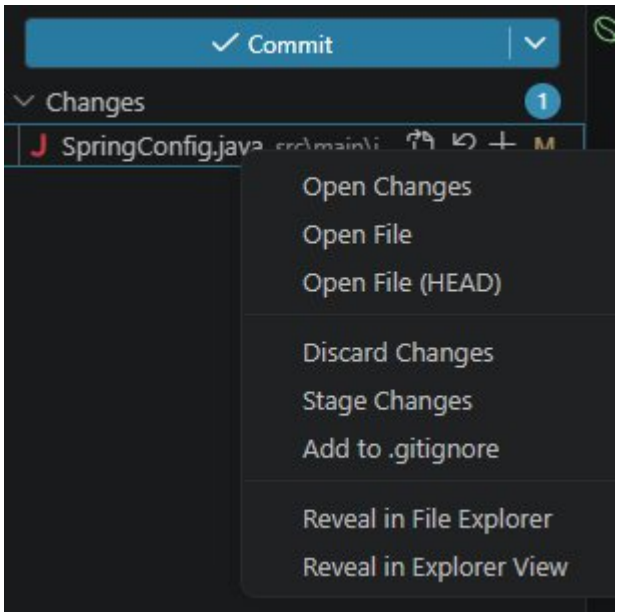
Когда мы щёлкаем по этой кнопке, открывается панель работы с Git. Она поделена на две основные части: CHANGES (изменения) и GRAPH (граф).

В верхней части, как следует из её заголовка, мы видим список файлов, имеющих изменения по сравнению с текущим коммитом:

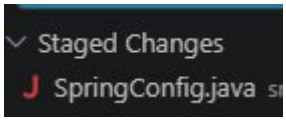


Изменения помечаются тем или иным символом в зависимости от их вида. Так, неотслеженный файл (unstaged) обозначается как U, изменённый (modified) как M, и так далее.

По умолчанию эти изменения не уходят в индекс автоматически. Чтобы добавить файл в индекс, посмотреть произведённые изменения и т.д., используется, например, контекстное меню:



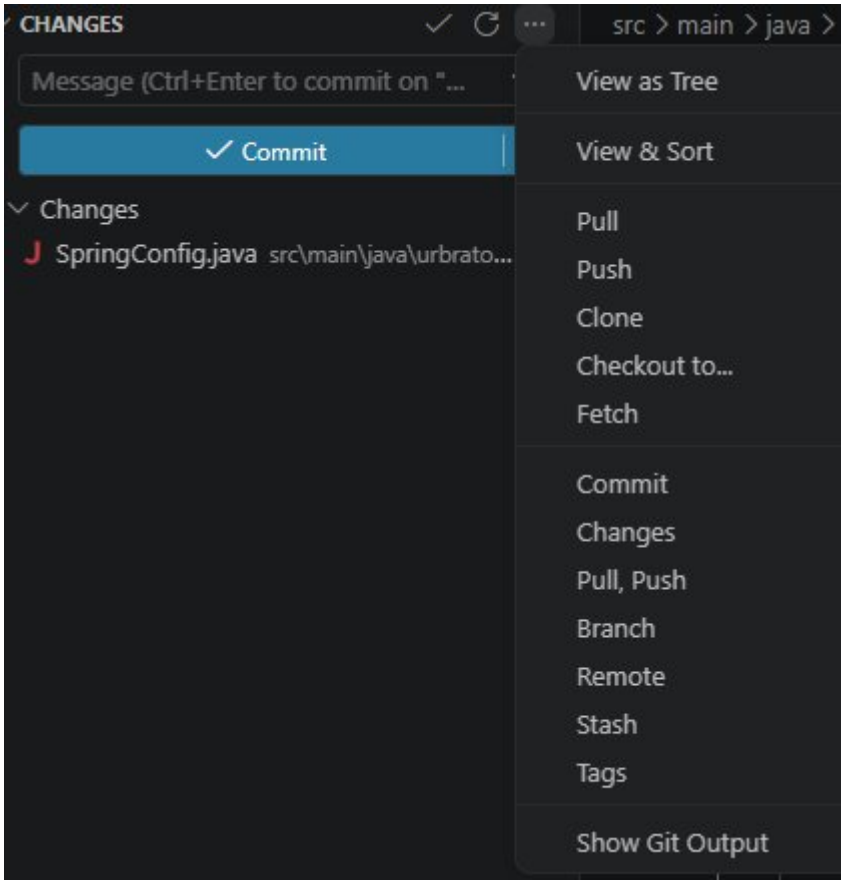
На самом деле, можно обойтись и без этого меню. Когда мы наводим мышь на то или иное изменение, в правой части списка появляются кнопки, при помощи которых можно, в частности, добавить изменение в индекс или отменить его (заменив на версию из последнего коммита). Когда индекс не пустой, кроме списка Changes появляется также список Staged Changes:



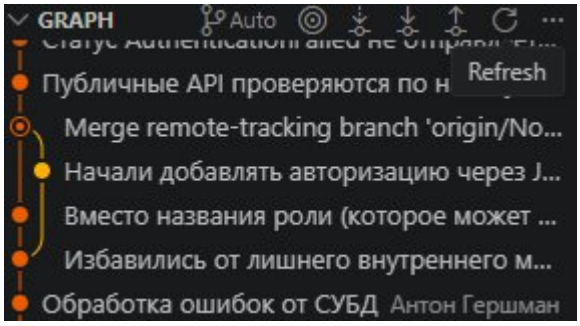
Точно так же уже добавленное в индекс изменение можно убрать из индекса, либо при помощи появляющейся кнопки, либо при помощи контекстного меню.

Когда мы готовы выполнить коммит, мы вводим сообщение коммита в текстовое поле над кнопкой Commit и нажимаем её. Но если внимательно посмотреть на эту кнопку, мы увидим, что она соединена с выпадающим списком. В нём мы можем выбрать варианты коммита: можно сделать простой коммит, можно выполнить исправляющий коммит (amend), а можем сразу после коммита выполнить пуш или синхронизацию (когда делается сначала пулл, а потом пуш). Синхронизацию одним нажатием выполнять не рекомендуется, если возможно, что в удалённом репозитории есть конфликтующие с вашими коммитами изменения. Для этого случая выполняйте пулл и пуш по отдельности.

Справа от заголовка CHANGES имеется кнопка, по которой открывается меню работы с Git, в котором содержатся практически все основные команды, включая работу с ветками и тайником:



В части GRAPH в графическом виде представлен журнал коммитов. Мы можем просматривать изменения, зафиксированные в каждом коммите, переходить между ними и выполнять другие действия:



Также в заголовке GRAPH имеются кнопки, позволяющие выполнить пулл, пуш, общую синхронизацию, перейти с ветки на ветку и т.д. Щелчок по кружку коммита открывает просмотр сделанных в коммите изменений прямо в редакторе, это очень удобная возможность.

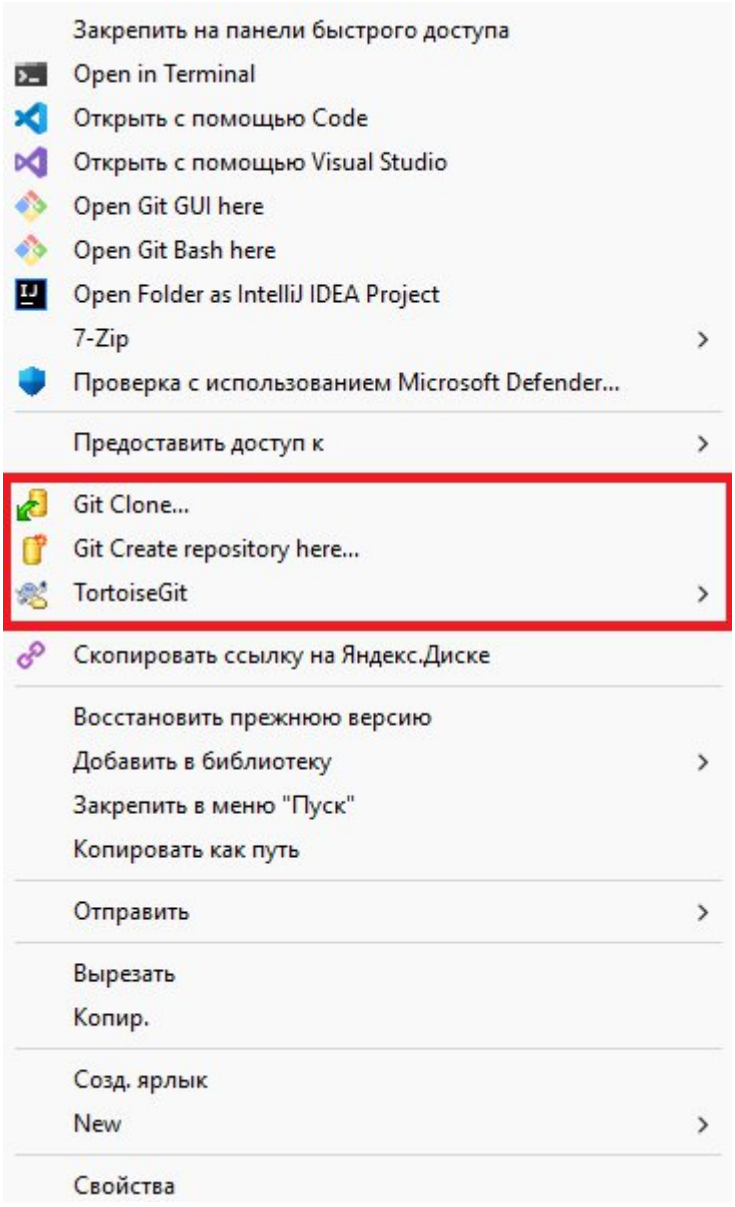
Обратите внимание, что когда мы нажимаем кнопку выбора ветки, список ветки выпадает не по месту щелчка, а опять же из верхней части окна Visual Studio Code. Если этого не заметить, может возникнуть ощущение, будто кнопка не работает.

§ 4.4. Tortoise Git for Windows

Говоря о графических клиентах Git, было бы неправильно обойти вниманием расширение Проводника Windows под названием Tortoise Git. Пользовательский интерфейс этого клиента практически совпадает с таковым для Tortoise SVN, более раннего клиента для Subversion, и реализован в виде расширения Проводника, добавляющего ряд пунктов в контекстное меню.

Это не просто бесплатный клиент. Немаловажен тот факт, что Tortoise Git – продукт с открытым исходным кодом, доступным на GitLab, распространяющийся по лицензии GNU GPL. Такая модель даёт возможность любому желающему сделать свой форк Tortoise Git и дополнить его разнообразными возможностями на свой вкус.

Если мы в Проводнике (или, к примеру, в Total Commander) щёлкнем правой кнопкой мыши на произвольной папке, мы увидим, что к контекстному меню теперь добавились несколько пунктов, которых не было до установки Tortoise Git:



Как мы видим, можно в одно действие создать репозиторий из выбранной папки, склонировать туда существующий облачный репозиторий, а также раскрыть более подробное меню Tortoise Git, содержащее более полный перечень команд. Заметим, что в Проводнике под Windows 11 по умолчанию пункты контекстного меню, добавленные Tortoise Git, скрыты, и чтобы получить к ним доступ, нужно сначала выбрать в контекстном меню пункт «Показать дополнительные параметры».

Если мы вызовем контекстное меню для папки, которая уже превращена в репозиторий, или для любой входящей в тот или иной репозиторий подпапки, мы увидим несколько другие пункты:

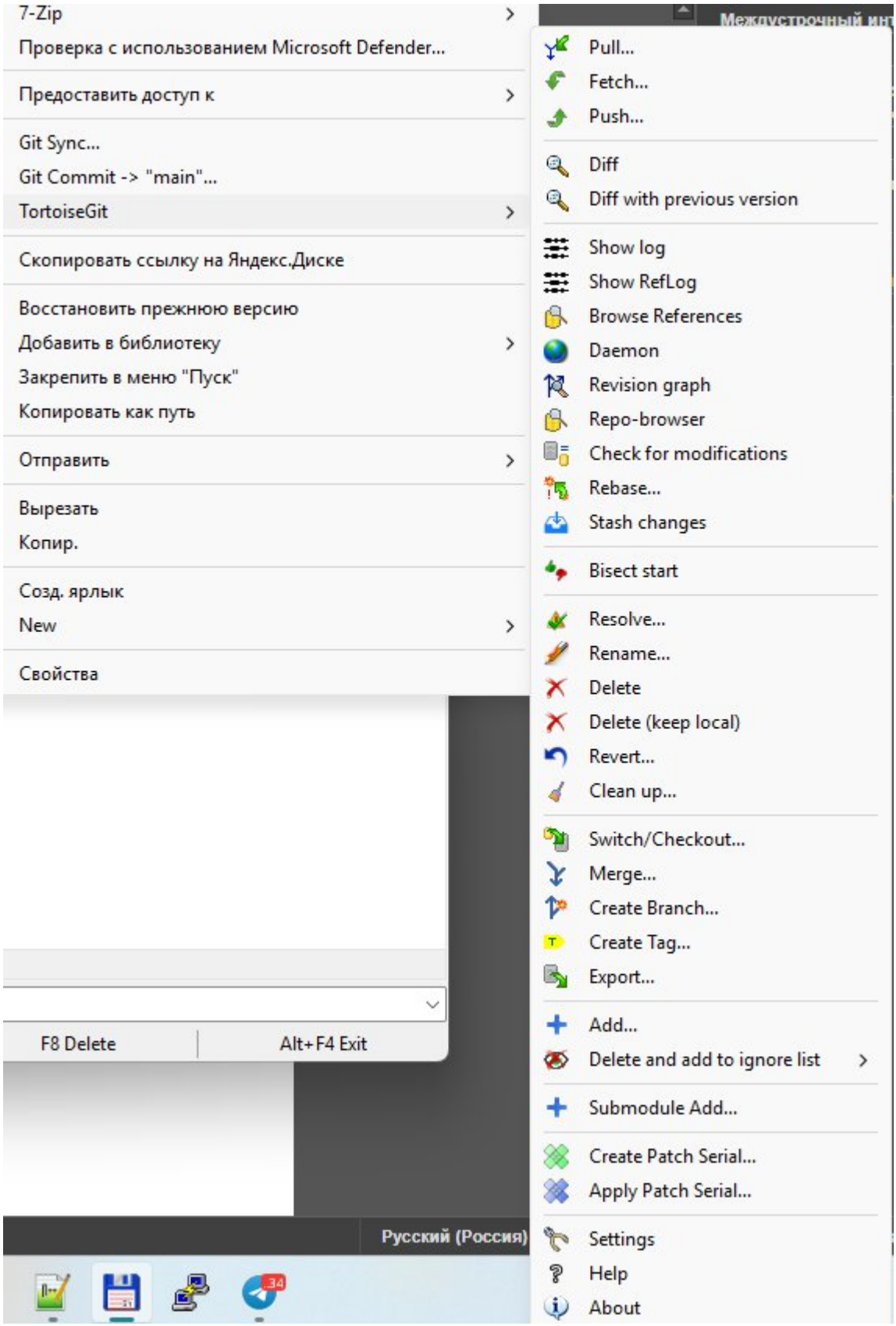
В самом деле, в таком случае создать в этой папке репозиторий или склонировать его в эту папку нельзя, зато можно выполнить коммит либо произвести синхронизацию репозитория с облачным сервисом (то есть, последовательно выполнить пулл и пуш).

Раскрыв меню Tortoise Git, мы увидим перечень почти всех возможностей Git.

Некоторые из присутствующих в этом меню возможностей мы даже не рассматривали в данном учебнике, ввиду того, что они используются довольно редко, а мы не ставим себе целью перечислить абсолютно всё, что можно сделать при помощи Git (да это и невозможно – как любой программный продукт, Git постоянно развивается). Однако если когда-нибудь какая-то из этих возможностей понадобится, всегда можно обратиться к документации.

Из минусов Tortoise Git можно отметить то, что поддержка работы с промежуточной стадией отслеженных, но не закоммиченных изменений до версии 2.10 находилась в состоянии разработки. Сейчас можно при помощи чекбоксов в окне коммита можно выбрать, что попадёт в индекс и затем в коммит. Однако для сложных сценариев работы с индексом Tortoise Git пока не готов. В частности, бывают непредвиденные результаты при частичном выводе изменений из индекса.

Вместе с тем в целом это довольно удобный и наглядный инструмент, особенно для сравнения коммитов между собой и с рабочим состоянием проекта, для просмотра истории в графическом виде и для разрешения конфликтов.



Контекстное меню Tortoise Git

На этом наш учебник завершается. За эти страницы мы прошли путь от первых команд в терминале до профессиональных рабочих процессов: ветвления, проверки кода, переписывания истории и защиты секретов. Надеемся, что Git перестал быть для вас «чёрным ящиком» и стал понятным, надёжным инструментом, который экономит время и нервы.

Удачных коммитов и чистых историй!

Благодарности

Благодарю Господа (и это не всуе) за то, что дал мне силы и усидчивости завершить начатое (и, разумеется, за прочие бесчисленные блага).

Благодарю создателей Git, в частности, автора системы Линуса Торвальдса и координатора проекта Джунио Хамано, без которых и речи бы не было.

Благодарю крутейшего энтузиаста бесплатного выращивания айтишников Куинси Ларсона за его бесценный труд, которому этот учебник обязан, наверное, минимум на две трети, а в части методологии и на четыре пятых.

Благодарю бесчисленных авторов Хабра и прочих ресурсов, позволяющих айтишникам учиться друг у друга.

Благодарю сердечно, неустанно и восхищённо мою семью за бесконечное терпение, позволившее мне выделить достаточно времени для написания этого учебника, и добровольный труд в качестве бета-читателей.

С уважением и глубокой признательностью благодарю коллег, также взявших на себя задачу проверки данного учебника на себе.

Наконец, горячо благодарю QWEN AI за оформление, неоценимую помощь в редактуре и корректуре, полезные замечания и правки.

Ответы к вопросам для самопроверки

§ 1.1: 1.2, 2.3, 3.4

§ 1.2: 1.1, 2.1, 3.3

§ 1.3: 1.2, 2.3, 3.2

§ 1.4: 1.3, 2.3, 3.4

§ 1.5: 1.2, 2.4, 3.3

§ 1.6: 1.3, 2.4, 3.2

§ 1.7: 1.2

§ 1.8: 1.2, 2.2, 3.1

§ 1.9: 1.2, 2.3, 3.4

§ 1.10: 1.2, 2.4, 3.2

§ 1.11: 1.2, 2.2, 3.4

§ 2.1 1.2, 2.2

§ 2.2: 1.3, 2.3, 3.(3,2,1,4)

§ 2.3: 1.2, 2.3, 3.2

§ 2.4: 1.(2,4), 2.2

§ 2.5: 1.2

§ 2.6: 1.2, 2.3