|  |  |
| --- | --- |
| Логотип | Государственное профессиональное образовательное автономное учреждение Ярославской области  «Ярославский промышленно-экономический колледж  им. Н.П. Пастухова» |
|  | |
| **Методические указания по работе в системе контроля версий GIT*****для специальности******09.02.07 Информационные системы и программирование*** 2021 | |

# **ПРЕДИСЛОВИЕ**

Данные методические указания предназначены для студентов специальности 09.02.07 «Информационные системы и программирование».

Методические указания содержат:

* требования к знаниям и умениям студентов;
* правила выполнения лабораторных работ (практических заданий);
* содержание лабораторных работ (практических заданий);
* контрольные вопросы;
* список использованной литературы;

В результате выполнения данного курса работ студент должен:

*Профессиональные компетенции*

ПК 2.1. Разрабатывать требования к программным модулям на основе анализа проектной и технической документации на предмет взаимодействия компонент.

ПК 2.2. Выполнять интеграцию модулей в программное обеспечение.

ПК 2.3. Выполнять отладку программного модуля с использованием специализированных программных средств.

ПК 2.4. Осуществлять разработку тестовых наборов и тестовых сценариев для программного обеспечения.

ПК 2.5. Производить инспектирование компонент программного обеспечения на предмет соответствия стандартам кодирования.

*Общие компетенции*

ОК 1. Выбирать способы решения задач профессиональной деятельности, применительно к различным контекстам.

ОК 2. Осуществлять поиск, анализ и интерпретацию информации, необходимой для выполнения задач профессиональной деятельности.

ОК 3. Планировать и реализовывать собственное профессиональное и личностное развитие.

ОК 4. Работать в коллективе и команде, эффективно взаимодействовать с коллегами, руководством, клиентами.

ОК 5. Осуществлять устную и письменную коммуникацию на государственном языке с учетом особенностей социального и культурного контекста.

ОК 6. Проявлять гражданско-патриотическую позицию, демонстрировать осознанное поведение на основе традиционных общечеловеческих ценностей.

ОК 7. Содействовать сохранению окружающей среды, ресурсосбережению, эффективно действовать в чрезвычайных ситуациях.

ОК 8. Использовать средства физической культуры для сохранения и укрепления здоровья в процессе профессиональной деятельности и поддержания необходимого уровня физической подготовленности.

ОК 9. Использовать информационные технологии в профессиональной деятельности.

ОК 10. Пользоваться профессиональной документацией на государственном и иностранном языке.

ОК 11. Планировать предпринимательскую деятельность в профессиональной сфере.

|  |  |  |
| --- | --- | --- |
| *Код*  *ПК, ОК* | *Умения* | *Знания* |
| ОК 1, 2, 3, 4, 5, 6, 7,8,9,10  ПК 2.1., 2.2., 2.3., 2.4., 2.5., | У1. использовать выбранную систему контроля версий;  У2. использовать методы для получения кода с заданной функциональностью и степенью качества | З1. модели процесса разработки программного обеспечения;  З2. основные принципы процесса разработки программного обеспечения;  З3. основные подходы к интегрированию программных модулей;  З4. основы верификации и аттестации программного обеспечения |

# **ПРАВИЛА ВЫПОЛНЕНИЯ ПРАКТИЧЕСКИХ ЗАНЯТИЙ**

* Лабораторные (практические) занятия проводятся в компьютерном кабинете
* Во время занятия в кабинете (лаборатории) обязательно присутствие преподавателя или лаборанта (техника).
* За каждым студентом закрепляется индивидуальное рабочее место (иное).
* Поведение студентов во время занятий регламентируется правилами техники безопасности.
* Первичный инструктаж по ТБ проводится в начале учебного года (семестра), с составлением контрольного листа на каждую студенческую группу, преподавателем, проводящим занятия.
* Текущий инструктаж по ТБ проводится перед каждым занятием.
* После выполнения всего курса работ обучающиеся предоставляют преподавателю отчет, оформленный в соответствии с требованиями к отчету.

# **ТРЕБОВАНИЯ К ОФОРМЛЕНИЮ ОТЧЁТА ПО УЧЕБНОЙ ПРАКТИКЕ**

1. Содержание отчета:

* Название и номер работы
* Цель работы
* Краткое описание выполнения всех заданий

1. Оформление отчета:

* Титульный лист
* Лист содержания
* Нумерация каждой страницы (начиная со следующего листа после содержания)
* В верхнем колонтитуле каждой страницы (начиная со следующей, после листа содержания):
* в левом углу – фамилия и инициалы студента
* в правом – шифр группы
* Требования к оформлению текста отчета:
* Поля: левое – 2 см, правое, верхнее, нижнее – по 1 см
* Шрифт: Times New Roman
* Размер шрифта – 14 пт
* Межстрочный интервал – 1,5
* Выравнивание – по ширине
* Отступ красной строки – 1,25 см

# **ТЕХНИКА БЕЗОПАСНОСТИ**

**Общие правила безопасности**

Сотрудникам и студенты, которым разрешена работа в ЛВС, выдается имя и пароль для работы в системе, а также, если это необходимо, для работы с дополнительными сервисами. Регистрация производится по письменному представлению заинтересованных подразделений администратором ЛВС колледжа.

При регистрации пользователи обязаны ознакомиться с данными правилами и расписаться в журнале по технике безопасности.

Зарегистрированным пользователям выделяется дисковое пространство и доступ к необходимому программному обеспечению в соответствии с текущими квотами.

При нарушении нормальной работы сети пользователи обязаны поставить в известность обслуживающий персонал.

**Пользователям запрещается:**

Самостоятельно производить установку, настройку, модификацию и тестирование сетевого аппаратного или программного обеспечения.

Использовать переносные носители информации. Работа с внешними носителями разрешается только преподавателю или сотруднику ЦИТ.

Изменять системные и сетевые настройки категорически запрещается.

Передавать по сети информацию, оскорбляющую честь и достоинство других абонентов сети, содержащую призывы к насилию, свержению существующего строя, разжиганию межнациональной розни, информацию в зашифрованном виде.

Использовать ресурсы сети для осуществления любого рода коммерческой деятельности.

Предпринимать какие-либо действия прямо или косвенно направленные на нарушение нормальной работы сетевого оборудования и разрушение общих информационных ресурсов.

Передавать кому-либо свой пароль, работать под чужим регистрационным именем, а также осуществлять любые действия, связанные с получением оных.

**Правила работы в компьютерных классах во время проведения занятий**

Пользователи обязаны придерживаться правил поведения в общественных местах. Количество пользователей не должно превышать количество рабочих мест за исключением специальных случаев (например, сдача зачетов и проч.).

В случае обнаружения неисправности компьютерного оборудования немедленно сообщить об этом преподавателю или сотруднику ЦИТ.

По окончанию работы корректно выйти из системы и сдать рабочее место преподавателю.

При самостоятельной работе по первому требованию преподавателя или сотрудника ЦИТ освободить компьютер.

Пользователям запрещается самостоятельно производить включение, выключение и перезагрузку компьютеров класса, подключение и отключение периферийных устройств.

Не допускается касание монитора руками, ручкой и др. предметами.

Пользователям запрещается находиться в классах в верхней одежде, располагать на рабочих столах шапки, сумки и пр.

Категорически запрещается играть в компьютерные игры, прослушивать музыку, просматривать фильмы, если это не предусмотрено планом урока.

Во время проведения занятий на преподавателей возлагается ответственность за обеспечение порядка в классе; при самостоятельных работах ответственность возлагается на преподавателя или сотрудника ЦИТ.

**Обязанности преподавателей**

Перед проведением практических занятий в компьютерном классе, преподаватель должен согласовать с Учебной частью и руководителем ЦИТ необходимый кабинет. При этом, во избежание срыва занятий, необходимо заранее определить с руководителем ЦИТ список программного обеспечения, которое потребуется для работы.

При необходимости выхода студентов в Интернет, следует подать заявку на внутреннем сайте [www.yxmt.local/staff](http://www.yxmt.local/staff)

При проведении занятий с использованием ресурсов Интернет, преподаватель должен отслеживать посещаемые Интернет-ресурсы студентами и пресекать попытки просмотра материала, содержание которого несовместимо с задачами воспитания и образования.

Во время занятий преподаватель должен находиться в компьютерном классе. В случае необходимости на время покинуть класс, преподаватель имеет право попросить техников или лаборантов ЦИТ присутствовать в классе вместо него.

Преподаватель, во время занятий, несет ответственность за сохранность оборудования и порядок в классе.

**Обязанности техников и лаборантов во время проведения занятий**

Обеспечение проведения занятий и самостоятельной работы пользователей в классах ЦИТ (включение ПК, присутствие в ЦИТ во время занятий для устранения возникающих проблем).

Установка и конфигурирование программного обеспечения, необходимого для проведения занятий.

Восстановление поврежденного системного и прикладного программного обеспечения, а также операционной системы.

Очистка компьютеров и ковриков для мышек от пыли, протирка поверхности мышек и клавиатуры чистящими салфетками, очистка поверхности монитора с помощью предназначенных для этого моющих средств с периодичностью раз в месяц или ранее по заявке преподавателей.

Получение со склада (или приобретение) расходных материалов, необходимых для эксплуатации компьютеров.

Данные правила являются обязательными для выполнения всеми пользователями. При нарушении правил, пользователь может быть лишен доступа к работе в ЛВС. При выполнении действий, влекущих за собой материальный ущерб пользователь несет ответственность, установленную администрацией.

# **КРИТЕРИИ ОЦЕНКИ**

Итоговой формой контроля знаний, умений и навыков учебной практики по получению первичных профессиональных умений и навыков, в том числе первичных умений и навыков научно-исследовательской деятельности является дифференцированный зачет.

Оценка результатов лабораторных занятий по МДК 02.01 проводится по следующим критериям:

Оценка «отлично» ставится, когда студентом:

* продемонстрирован высокий уровень общекультурных и общепрофессиональных компетенций;
* выполнен в срок и на высоком уровне весь намеченный объем работы в соответствии с заданием лабораторных занятий;
* проявлены самостоятельность, творческий подход и высокий уровень подготовки по вопросам профессиональной деятельности, самоорганизации;
* -оформлен отчет в соответствии с требованиями.

Оценка «хорошо» ставится, когда студентом:

* в целом продемонстрирована сформированность общекультурных и общепрофессиональных компетенций;
* выполнено полностью задание лабораторных работ, однако допущены незначительные недочеты при написании отчета, в основном технического характера.
* Оценка «удовлетворительно» ставится, когда студентом:
* продемонстрирована сформированность отдельных общекультурных и общепрофессиональных компетенций;
* допущены существенные недочеты в составлении отчета.

Оценка «неудовлетворительно» ставится, когда студентом:

* не продемонстрирована сформированность общекультурных и общепрофессиональных компетенций;
* не выполнено задание практики;
* студент представил небрежно оформленный отчет по лабораторным работам.

# **ПРАКТИЧЕСКИЕ РАБОТЫ**

**Практическая работа №1**

**Тема**

Создание репозитория в хостинге IT-проектов GITHUB.

**Цель работы**

Получить начальный опыт создания репозитория в хостинге IT-проектов GITHUB.

**Задание**

1. Изучите указанный теоретический материал.
2. *Зарегистрируйтесь на GITHUB.*
3. В хостинге IT-проектов GITHUB создайте приватный репозиторий.
4. *Оформите отчет.*

**Теоретический материал**

**О системе контроля версий**

Что такое «система контроля версий» и почему это важно? *Система контроля версий* — это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии. Для контроля версий файлов в этой книге в качестве примера будет использоваться исходный код программного обеспечения, хотя на самом деле вы можете использовать контроль версий практически для любых типов файлов.

Если вы графический или web-дизайнер и хотите сохранить каждую версию изображения или макета (скорее всего, захотите), система контроля версий (далее СКВ) — как раз то, что нужно. Она позволяет вернуть файлы к состоянию, в котором они были до изменений, вернуть проект к исходному состоянию, увидеть изменения, увидеть, кто последний менял что-то и вызвал проблему, кто поставил задачу и когда и многое другое. Использование СКВ также значит в целом, что, если вы сломали что-то или потеряли файлы, вы спокойно можете всё исправить. В дополнение ко всему вы получите всё это без каких-либо дополнительных усилий.

**Локальные системы контроля версий**

Многие люди в качестве метода контроля версий применяют копирование файлов в отдельную директорию (возможно даже, директорию с отметкой по времени, если они достаточно сообразительны). Данный подход очень распространён из-за его простоты, однако он невероятно сильно подвержен появлению ошибок. Можно легко забыть, в какой директории вы находитесь, и случайно изменить не тот файл или скопировать не те файлы, которые вы хотели. Для того, чтобы решить эту проблему, программисты давным-давно разработали локальные СКВ с простой базой данных, которая хранит записи о всех изменениях в файлах, осуществляя тем самым контроль ревизий.

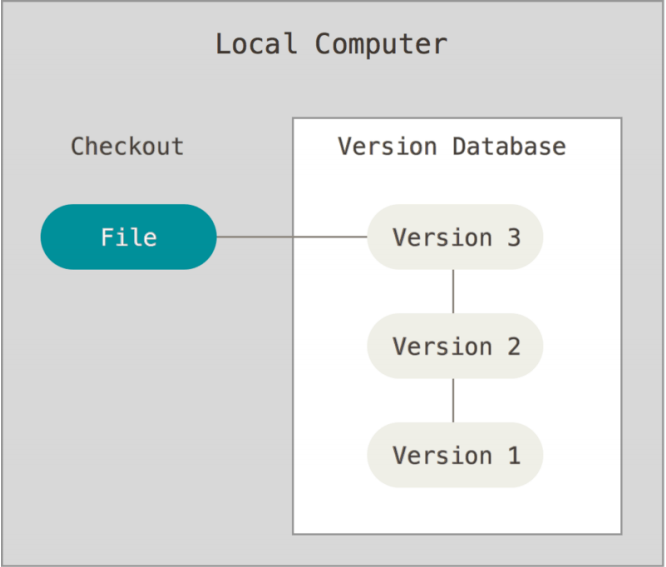


Рисунок 1. Локальный контроль версий.

Одной из популярных СКВ была система RCS, которая и сегодня распространяется со многими компьютерами. RCS хранит на диске наборы патчей (различий между файлами) в специальном формате, применяя которые она может воссоздавать состояние каждого файла в заданный момент времени.

**Централизованные системы контроля версий**

Следующая серьёзная проблема, с которой сталкиваются люди, — это необходимость взаимодействовать с другими разработчиками. Для того, чтобы разобраться с ней, были разработаны централизованные системы контроля версий (ЦСКВ). Такие системы, как CVS, Subversion и Perforce, используют единственный сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища. Применение ЦСКВ являлось стандартом на протяжении многих лет.

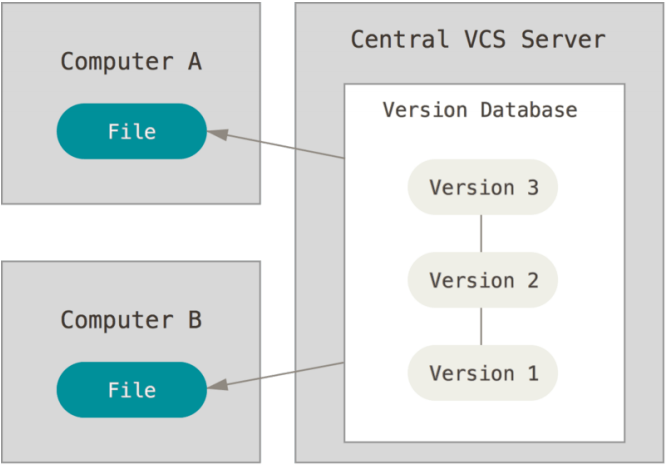


Рисунок 2. Централизованный контроль версий.

Такой подход имеет множество преимуществ, особенно перед локальными СКВ. Например, все разработчики проекта в определённой степени знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать, и гораздо проще администрировать ЦСКВ, чем оперировать локальными базами данных на каждом клиенте.

Несмотря на это, данный подход тоже имеет серьёзные минусы. Самый очевидный минус — это единая точка отказа, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого времени никто не сможет использовать контроль версий для сохранения изменений, над которыми работает, а также никто не сможет обмениваться этими изменениями с другими разработчиками. Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные бэкапы отсутствуют, вы потеряете всё — всю историю проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков. Локальные СКВ страдают от той же самой проблемы: когда вся история проекта хранится в одном месте, вы рискуете потерять всё.

**Распределённые системы контроля версий**

Здесь в игру вступают распределённые системы контроля версий (РСКВ). В РСКВ (таких как Git, Mercurial, Bazaar или Darcs) клиенты не просто скачивают снимок всех файлов (состояние файлов на определённый момент времени) — они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, умрёт, любой клиентский репозиторий может быть скопирован на 10 другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом всех данных.

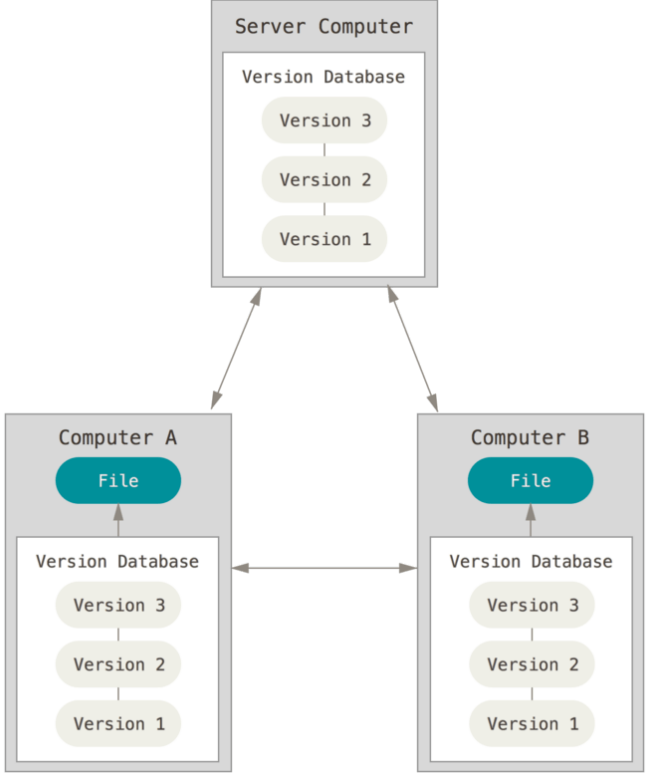


Рисунок 3. Распределённый контроль версий.

Более того, многие РСКВ могут одновременно взаимодействовать с несколькими удалёнными репозиториями, благодаря этому вы можете работать с различными группами людей, применяя различные подходы единовременно в рамках одного проекта. Это позволяет применять сразу несколько подходов в разработке, например, иерархические модели, что совершенно невозможно в централизованных системах.

**Основы Git**

Git хранит и использует информацию совсем иначе по сравнению с другими системами.

Основное отличие Git’а от любой другой СКВ — это подход Git’а к работе со своими данными. Концептуально, большинство других систем хранят информацию в виде списка изменений в файлах. Эти системы (CVS, Subversion, Perforce, Bazaar и т.д.) представляют хранимую информацию в виде набора файлов и изменений, сделанных в каждом файле, по времени (обычно это называют контролем версий, основанным на различиях).

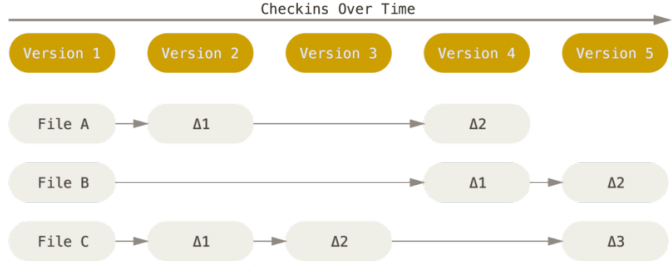


Рисунок 4. Хранение данных как набора изменений относительно первоначальной версии каждого из файлов.

Git не хранит и не обрабатывает данные таким способом. Вместо этого, подход Git’а к хранению данных больше похож на набор снимков миниатюрной файловой системы. Каждый раз, когда вы делаете *коммит*, то есть сохраняете состояние своего проекта в Git’е, система запоминает, как выглядит каждый файл в этот момент, и сохраняет ссылку на этот снимок. Для увеличения эффективности, если файлы не были изменены, Git не запоминает эти файлы вновь, а только создаёт ссылку на предыдущую версию идентичного файла, который уже сохранён. Git представляет свои данные как, скажем, поток снимков.

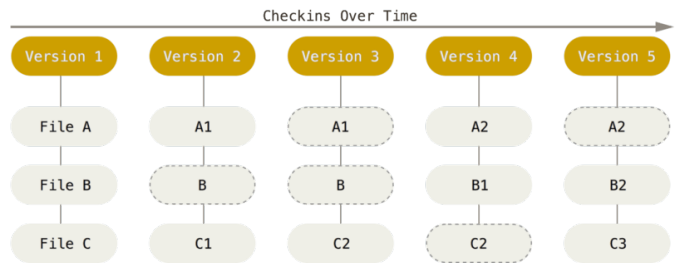


Рисунок 5. Хранение данных как снимков проекта во времени.

Это очень важное отличие между Git и почти любой другой СКВ. Git переосмысливает практически все аспекты контроля версий, которые были скопированы из предыдущего поколения большинством других систем. Это делает Git больше похожим на миниатюрную файловую систему с удивительно мощными утилитами, надстроенными над ней, нежели просто на СКВ.

Для работы большинства операций в Git достаточно локальных файлов и ресурсов — в основном, системе не нужна никакая информация с других компьютеров в вашей сети. История проекта хранится прямо на вашем локальном диске, большинство операций кажутся чуть ли не мгновенными.

Для примера, чтобы посмотреть историю проекта, Git’у не нужно соединяться с сервером для её получения и отображения — система просто считывает данные напрямую из локальной базы данных. Это означает, что вы увидите историю проекта практически моментально. Если вам необходимо посмотреть изменения, сделанные между текущей версией файла и версией, созданной месяц назад, Git может найти файл месячной давности и локально вычислить изменения, вместо того, чтобы запрашивать удалённый сервер выполнить эту операцию, либо вместо получения старой версии файла с сервера и выполнения операции локально.

Это также означает, что есть лишь небольшое количество действий, которые вы не сможете выполнить, если вы находитесь оффлайн или не имеете доступа к VPN в данный момент. Если вы в самолёте или в поезде и хотите немного поработать, вы сможете создавать коммиты без каких-либо проблем (в вашу локальную копию, помните?): когда будет возможность подключиться к сети, все изменения можно будет синхронизировать. Если вы ушли домой и не можете подключиться через VPN, вы всё равно сможете работать. Добиться такого же поведения во многих других системах либо очень сложно, либо вовсе невозможно.

В Git’е для всего вычисляется хеш-сумма, и только потом происходит сохранение. В дальнейшем обращение к сохранённым объектам происходит по этой хеш-сумме. Это значит, что невозможно изменить содержимое файла или директории так, чтобы Git не узнал об этом. Данная функциональность встроена в Git на низком уровне и является неотъемлемой частью его философии. Вы не потеряете информацию во время её передачи и не получите повреждённый файл без ведома Git. Механизм, которым пользуется Git при вычислении хеш-сумм, называется SHA-1 хеш.

Вы будете постоянно встречать хеши в Git’е, потому что он использует их повсеместно. На самом деле, Git сохраняет все объекты в свою базу данных не по имени, а по хеш-сумме содержимого объекта.

Когда вы производите какие-либо действия в Git, практически все из них только добавляют новые данные в базу Git. Очень сложно заставить систему удалить данные либо сделать что-то, что нельзя впоследствии отменить. Как и в любой другой СКВ, вы можете потерять или испортить свои изменения, пока они не зафиксированы, но после того, как вы зафиксируете снимок в Git, будет очень сложно что-либо потерять, особенно, если вы регулярно синхронизируете свою базу с другим репозиторием.

Всё это превращает использование Git в одно удовольствие, потому что мы знаем, что можем экспериментировать, не боясь серьёзных проблем. Для более глубокого понимания того, как Git хранит свои данные и как вы можете восстановить данные, которые кажутся утерянными, см. Операции отмены.

**Три состояния**

У Git’а есть три основных состояния, в которых могут находиться ваши файлы: ***зафиксированное*** (committed), ***изменённое*** (modified) и ***подготовленное*** (staged).

\* Зафиксированный значит, что файл уже сохранён в вашей локальной базе.

\* К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы.

\* Подготовленные файлы — это изменённые файлы, отмеченные для включения в следующий коммит.

Мы подошли к трём основным секциям проекта Git: Git-директория (Git directory), рабочая директория (working directory) и область подготовленных файлов (staging area).

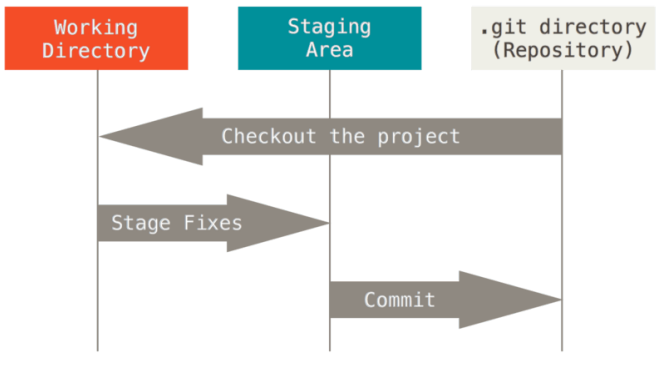


Рисунок 6. Рабочая директория, область подготовленных файлов и директория Git.

*Git-директория* — это то место, где Git хранит метаданные и базу объектов вашего проекта. Это самая важная часть Git, и это та часть, которая копируется при клонировании репозитория с другого компьютера.

Рабочая директория является снимком версии проекта. Файлы распаковываются из сжатой базы данных в Git-директории и располагаются на диске, для того чтобы их можно было изменять и использовать.

*Область подготовленных файлов* — это файл, обычно располагающийся в вашей Git-директории, в нём содержится информация о том, какие изменения попадут в следующий коммит. Эту область ещё называют “индекс”, однако называть её stage-область также общепринято. Базовый подход в работе с Git выглядит так:

1. Вы изменяете файлы в вашей рабочей директории.

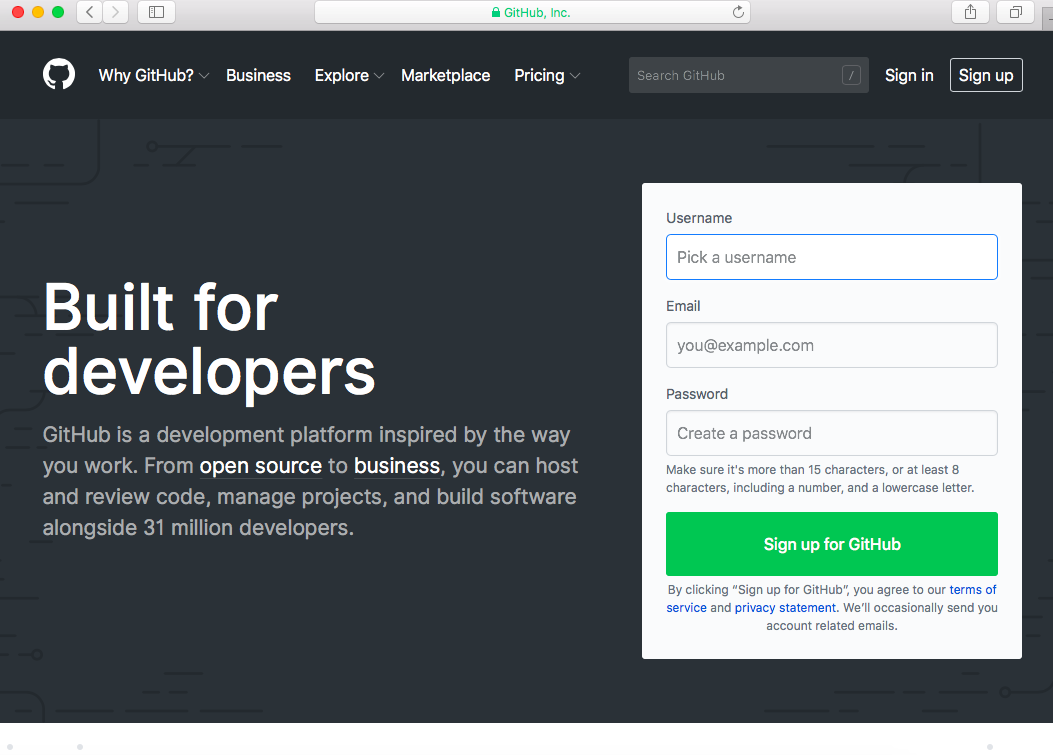
2. Вы выборочно добавляете в индекс только те изменения, которые должны попасть в следующий коммит, добавляя тем самым снимки только этих изменений в область подготовленных файлов.

3. Когда вы делаете коммит, используются файлы из индекса как есть, и этот снимок сохраняется в вашу Git-директорию.

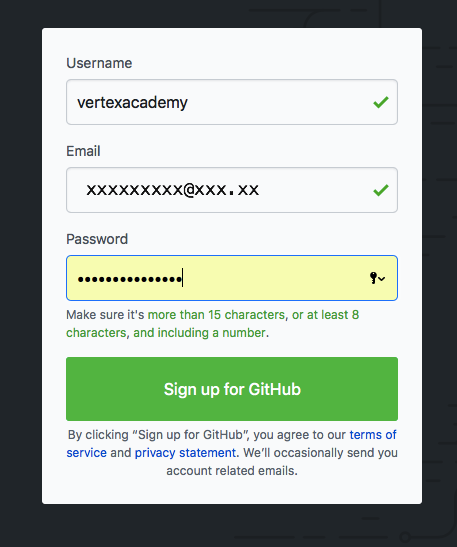
Если определённая версия файла есть в Git-директории, эта версия считается зафиксированной. Если версия файла изменена и добавлена в индекс, значит, она подготовлена. И если файл был изменён с момента последнего распаковывания из репозитория, но не был добавлен в индекс, он считается изменённым.

**Пояснение к работе**

**Регистрация на GitHub**



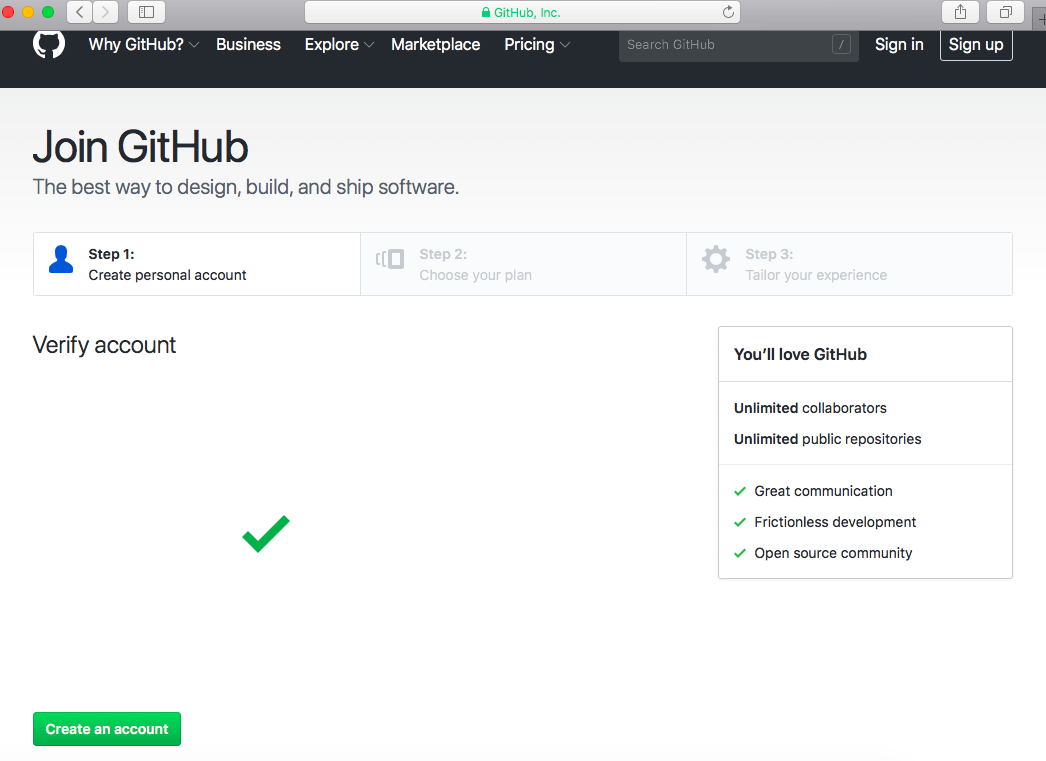
Итак, зарегистрироваться на GitHub очень просто: надо ввести имя, почту и придумать пароль. Например:



Имя, конечно же должно быть уникальным. Если имя уникальное, Вы увидите зеленую галочку справа (как на картинке).

Почта тоже должна быть уникальной. Нельзя зарегистрировать два аккаунта на одну и ту же почту.

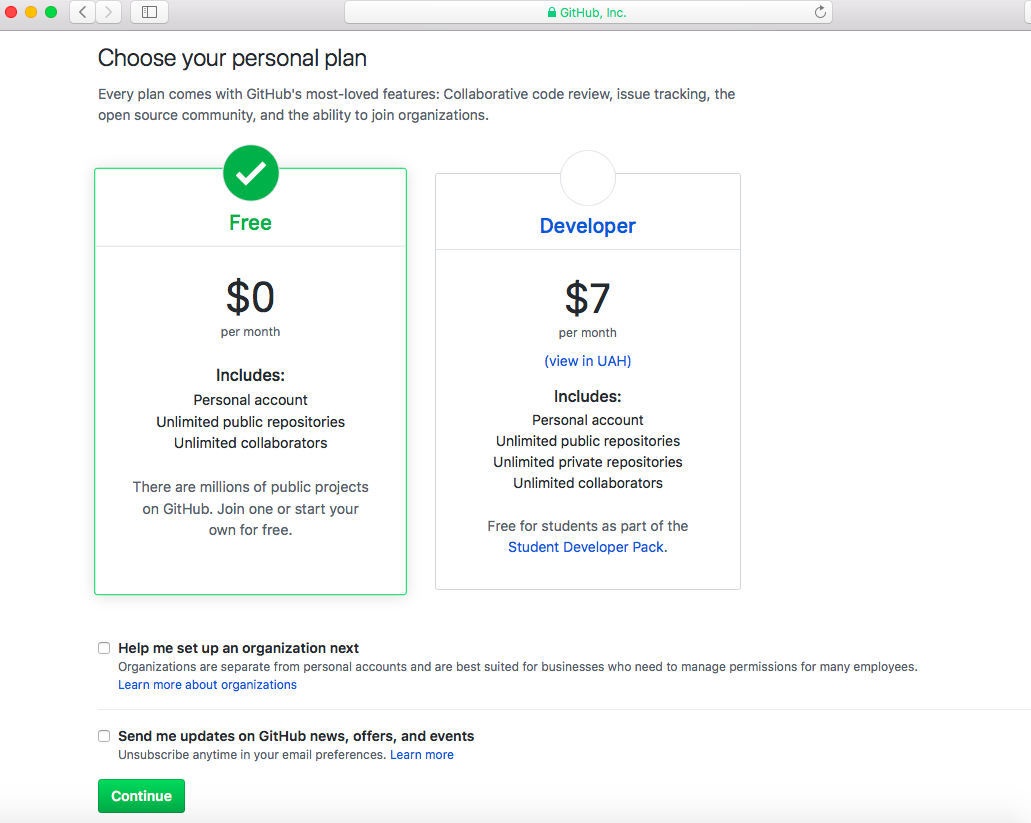
После этого шага, Вы попадете на такую страницу:



После этого нажимаем на зеленую кнопку внизу - "Create an account" ("создать аккаунт").



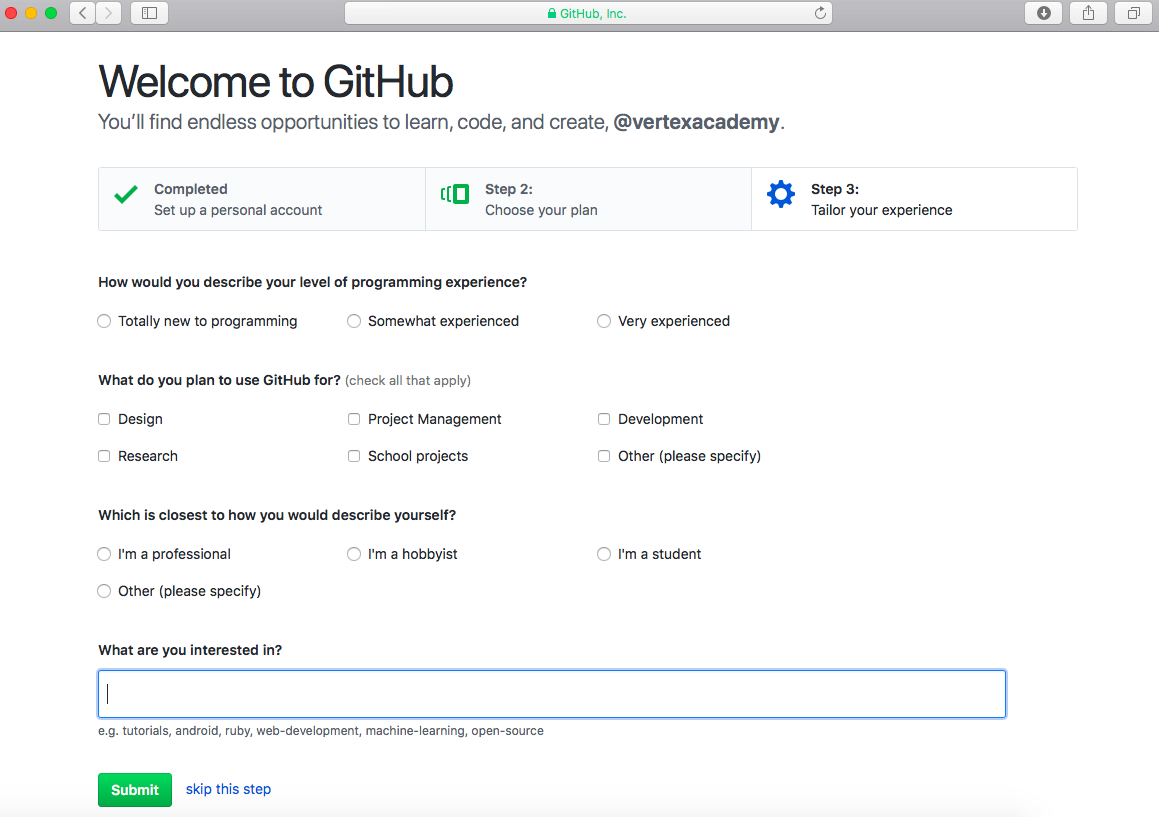
Появится окно, где Вы можете выбрать тарифный план. Первый -**бесплатный**, которым мы и будем пользоваться.



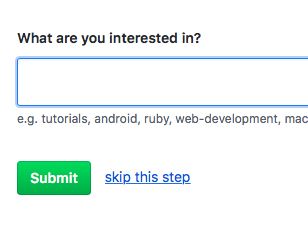
* Если Вы хотите неограниченное место для своих проектов, но не хотите, чтобы их кто-то видел, например, если Вы пишете проекты для работодателя - Вам придётся переключиться на платный аккаунт.

Жмём на зеленую кнопку "Continue".

Теперь мы на последнем шаге - шаге 3. Он называется "Tailor your experience" - тут GitHub спрашивает Вас, кто Вы и зачем хотите использовать GitHub. Этот шаг можно пропустить.

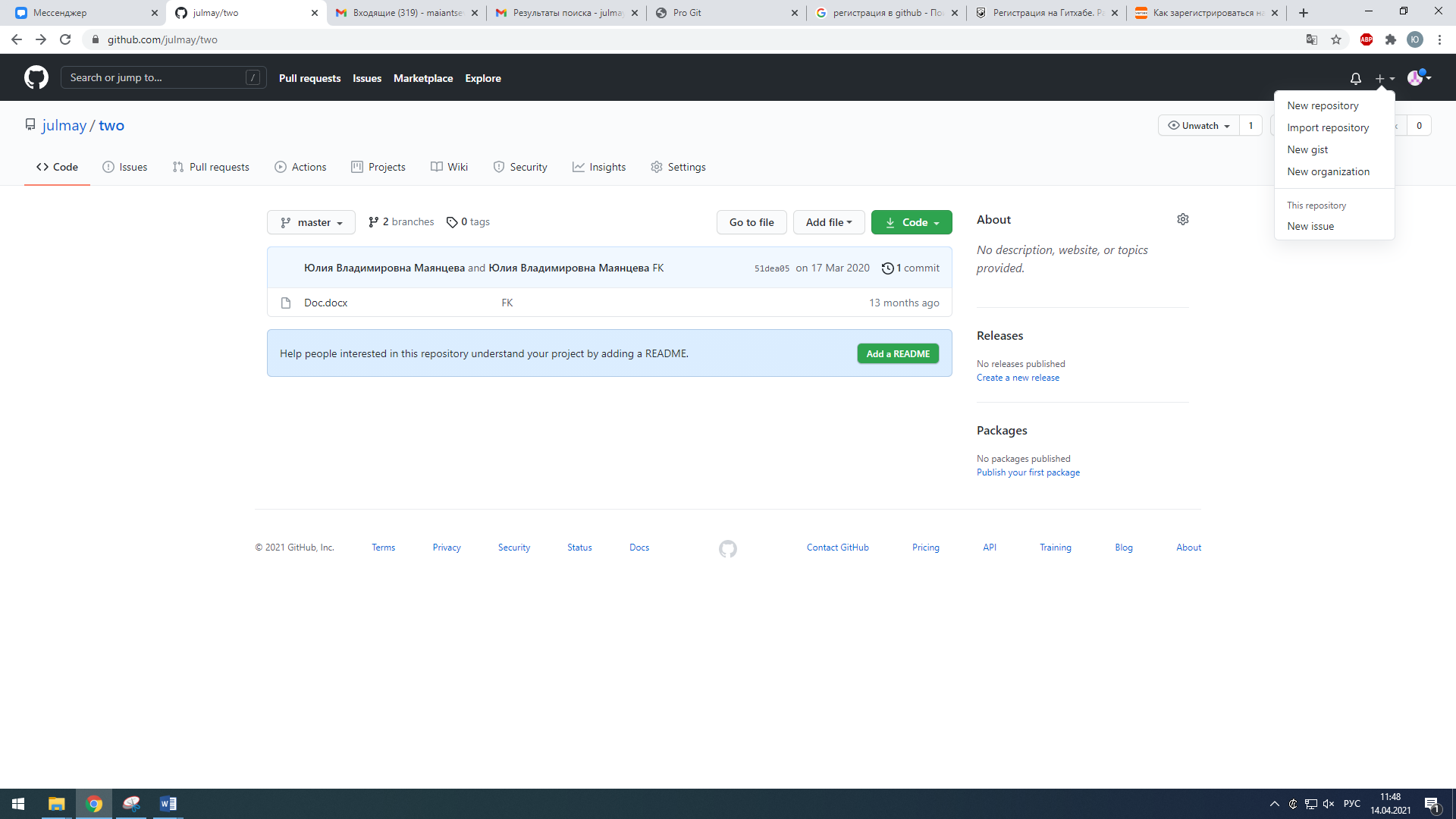


Чтобы пропустить этот шаг, жмём "**skip this step**" - то, что написано маленькими буквами рядом с зеленой кнопкой "**Submit**".

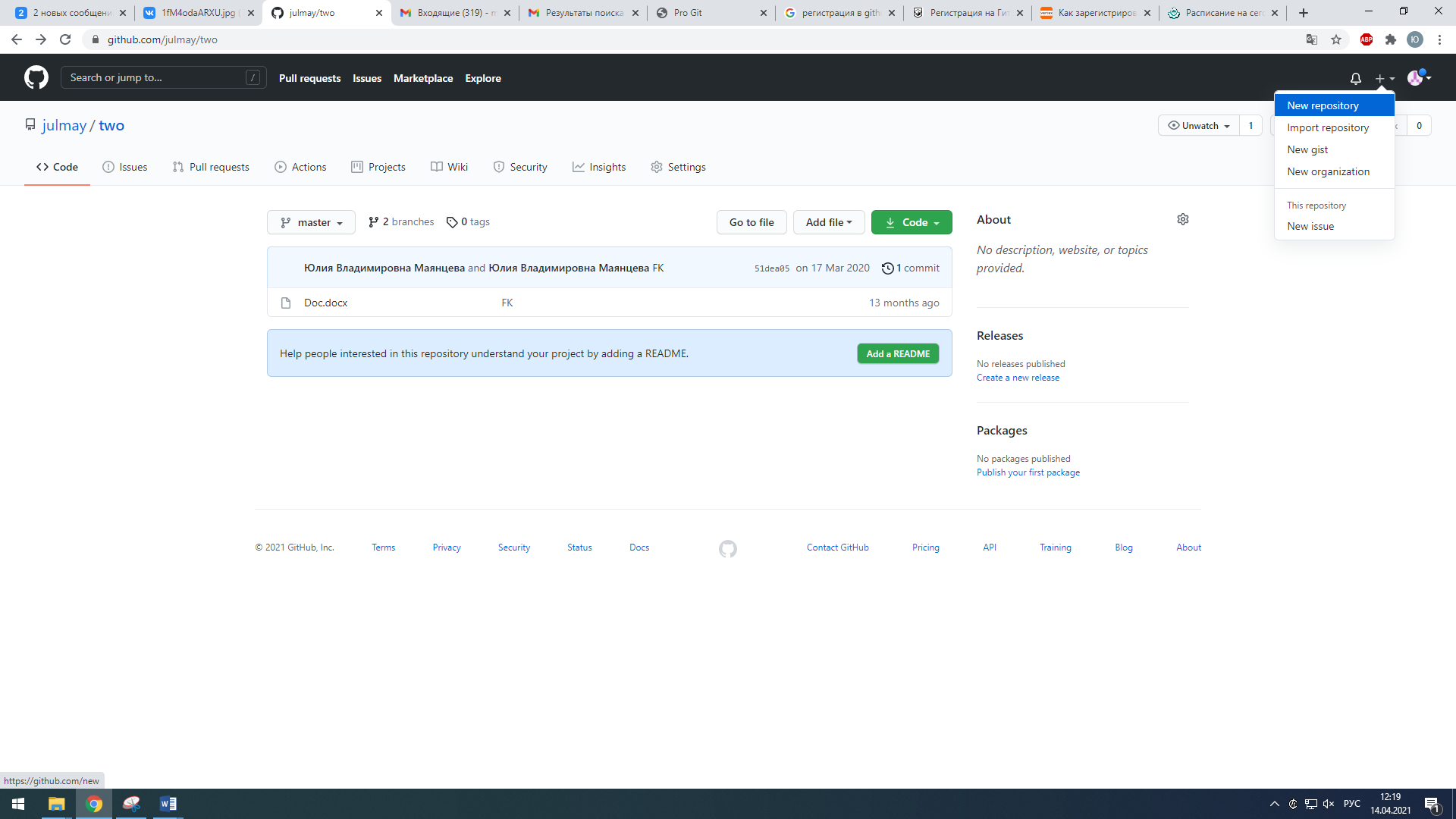


Поздравляем! Вы зарегистрировались на **GitHub. Теперь создадим репозиторий.**

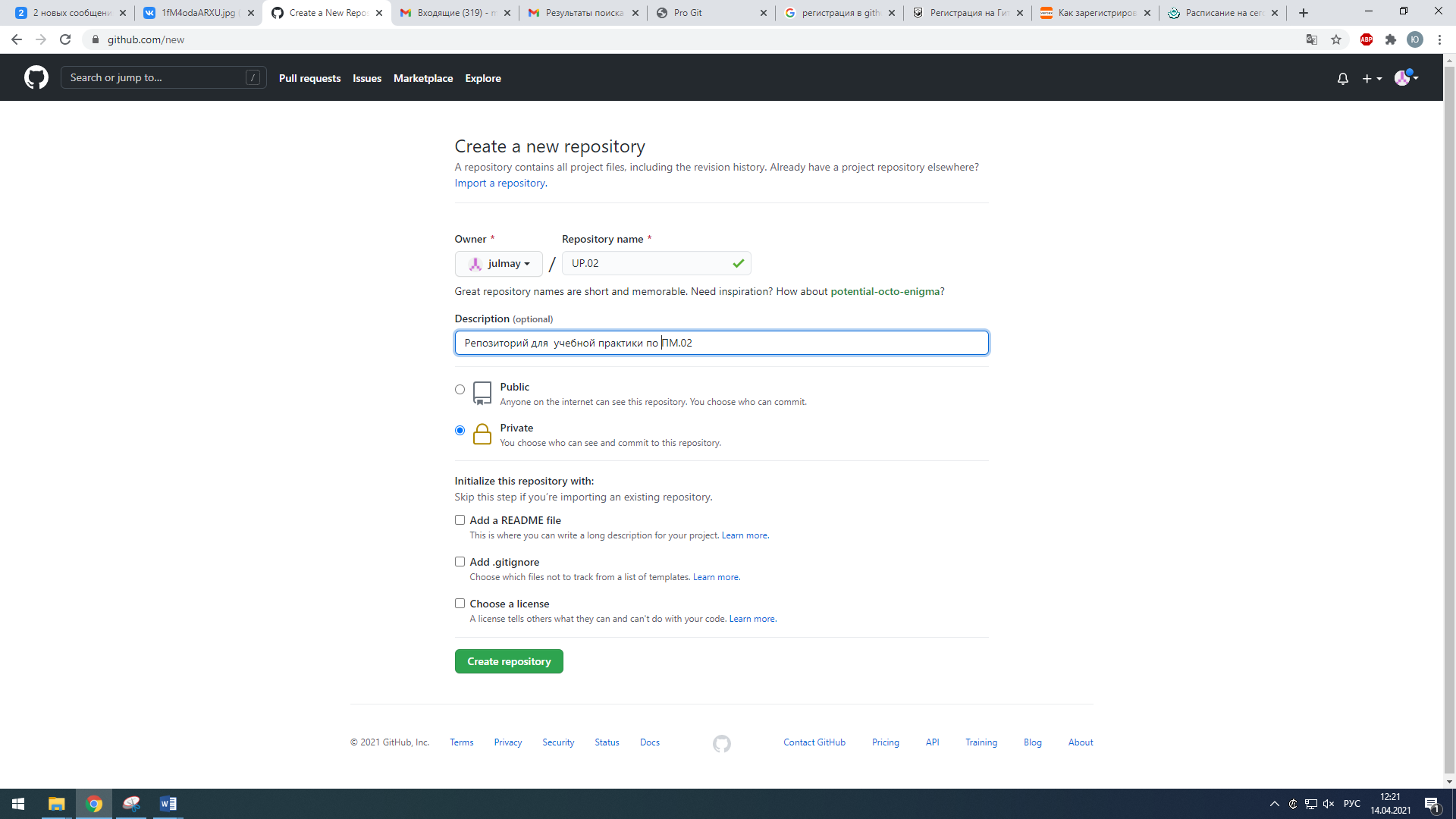
**Шаг 1.** Нажмите на «**+**» в правой верхней части окна.



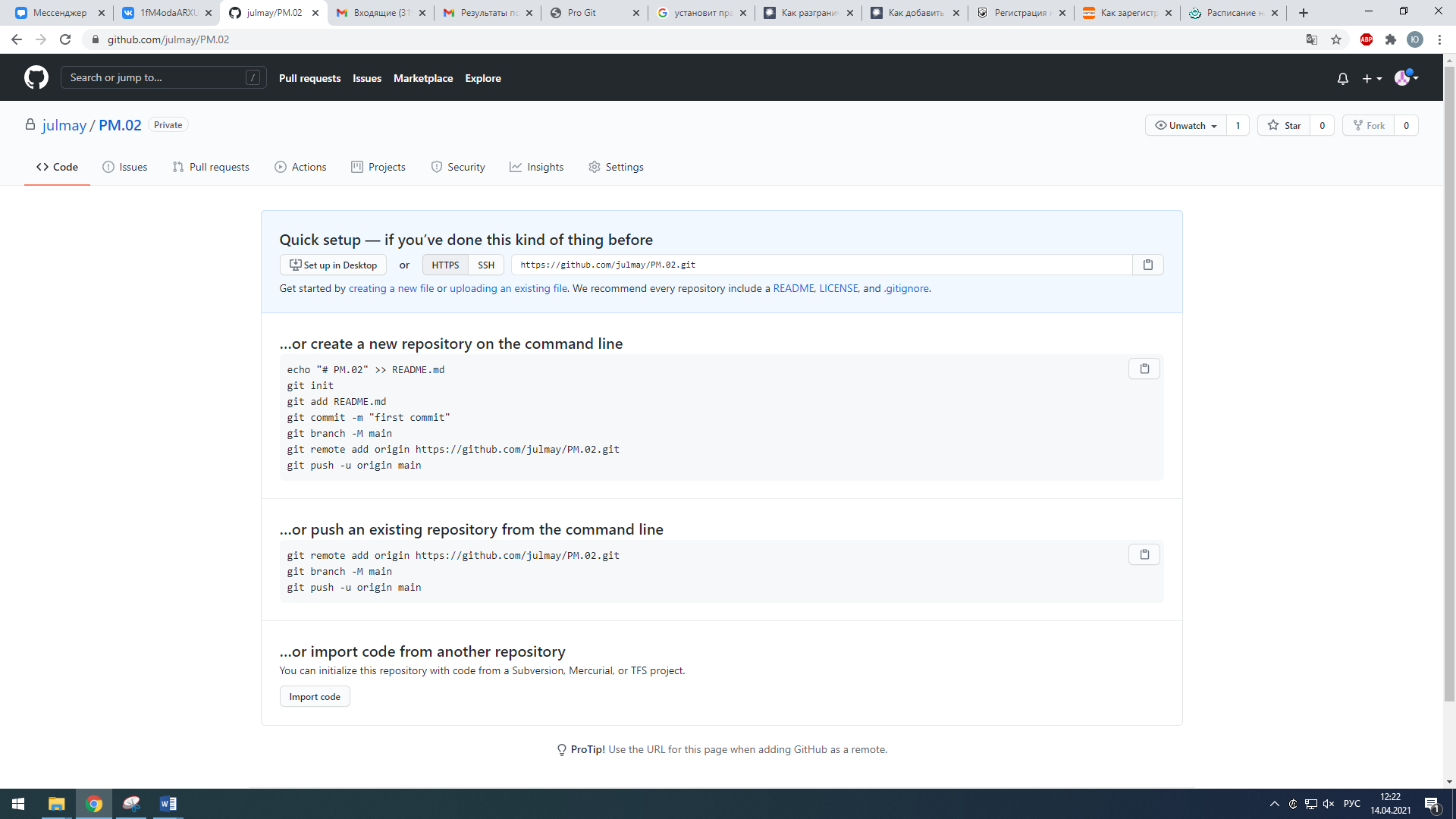
**Шаг 2.** Выберите **New repository**



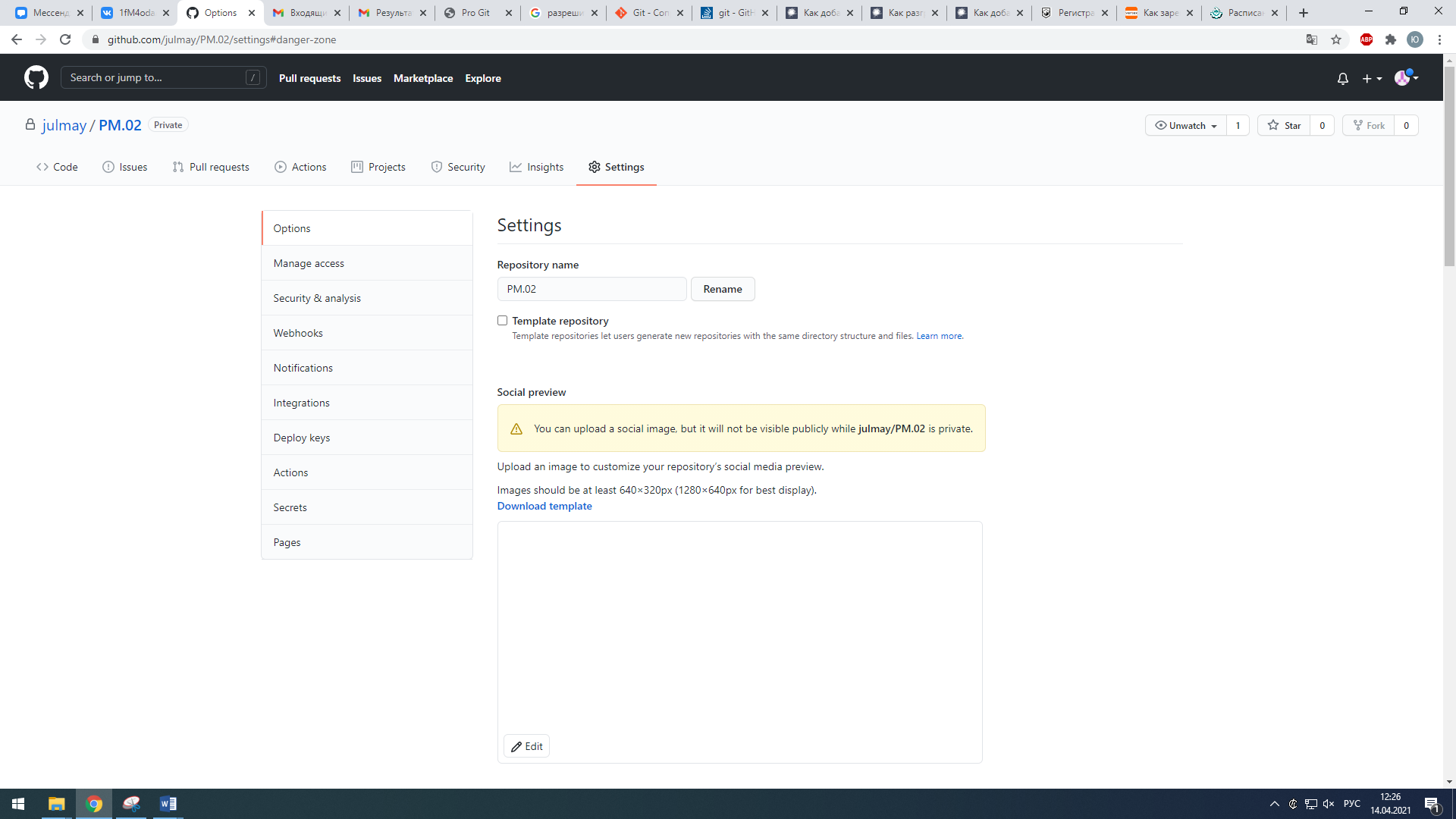
**Шаг 3.** В поле **Repository** name укажите имя репозитория в формате UP.номер\_модуля (например, UP.02). Выберите тип репозитория **Private**. И нажмите **Create repository**



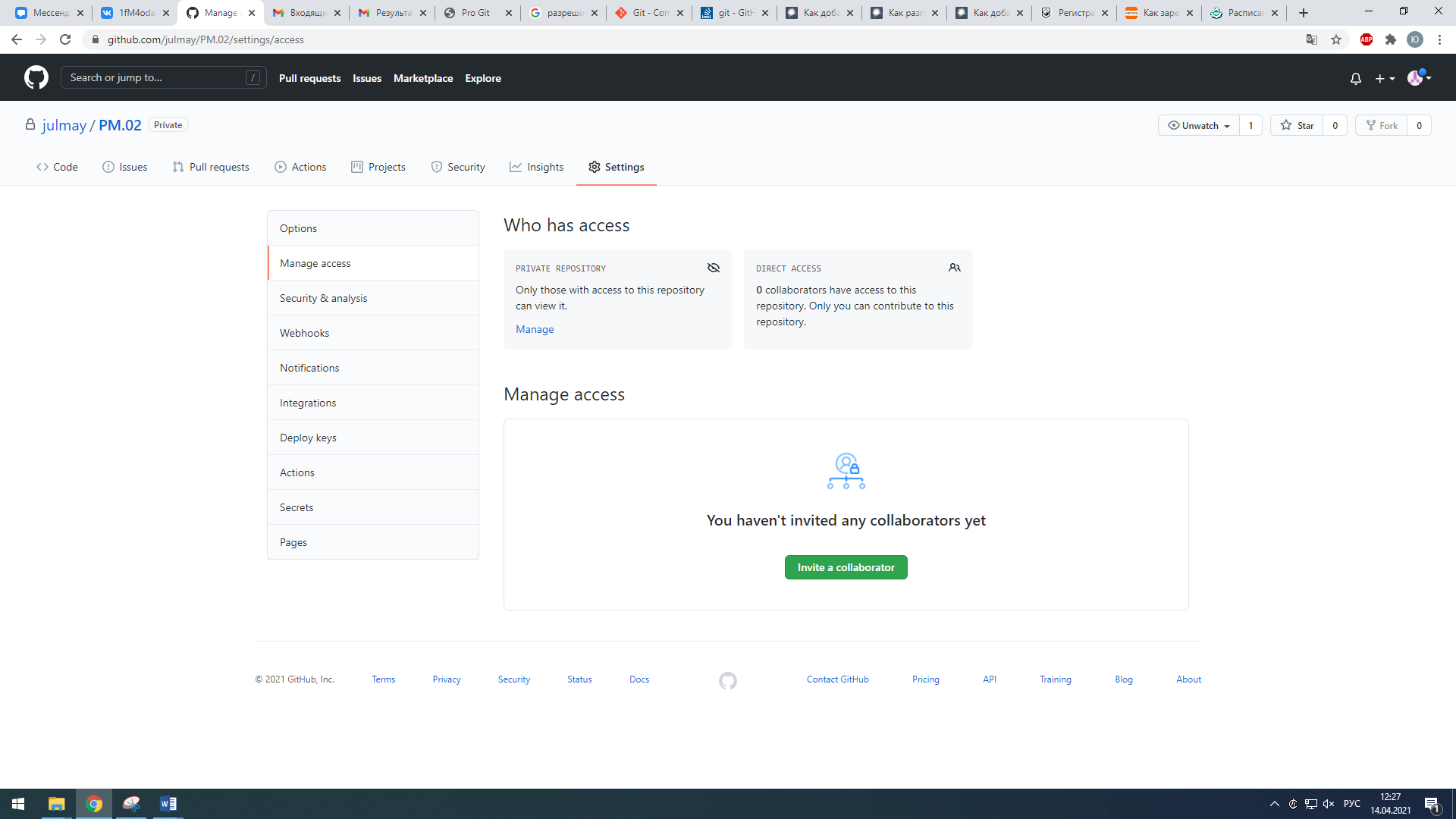
**Шаг 3.** На экране вы увидите:



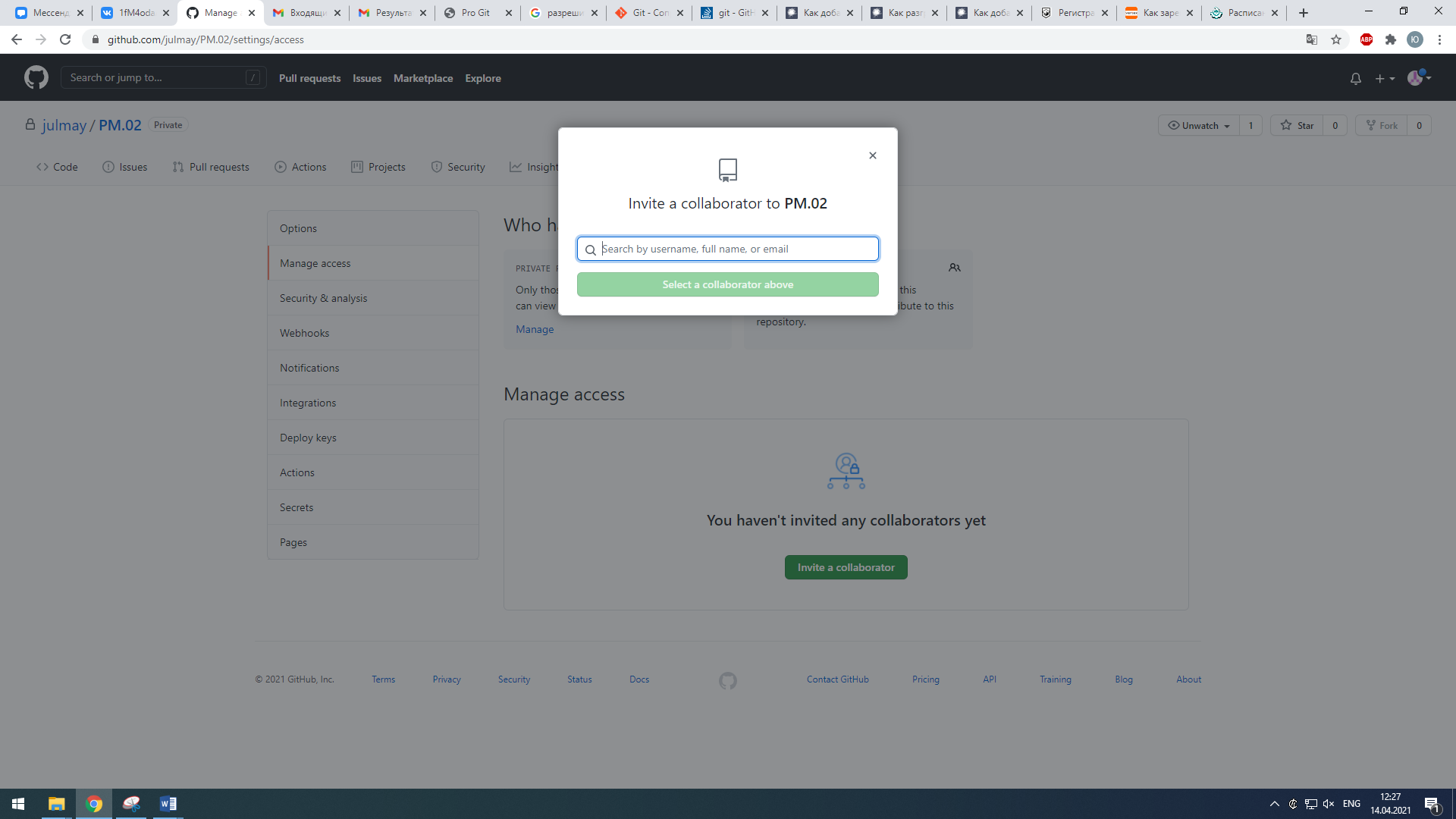
**Шаг 4.** Перейдем в раздел **Settings**



**Шаг 5.** В меню выберем **Manage access** и нажмем кнопку **Invite a collaborator**.



**Шаг 5.** В открывшемся диалоговом окне в строке поиска вписываем julmay и жмем на зеленую кнопку.



**Выполнение работы**

1. *Зарегистрируйтесь хостинге IT-проектов GITHUB (https://github.com/)*
2. *Скачайте задание из репозитория …*
3. Создайте приватный репозиторий с названием учебной практики (например, «UP\_02»).
4. Установите права доступа для учетной записи julmay (только для скачивания файлов).
5. *Оформите отчет.*

***Содержание отчета***

*Отчет по практической работе должен содержать:*

1. *Цель работы.*
2. *Раздел с заданием, в котором расписывается о том, каким образом студент выполнил его, с использованием каких средств, инструментов и технологий. Представленная информация подтверждается скриншотами экрана, файлами, полученными в ходе выполнения практической работы.*
3. *Ответы на контрольные вопросы.*

***Контрольные вопросы***

1. *Что такое «система контроля версий» (далее СКВ)?*
2. *В чем заключается экономия времени при использовании системы контроля версий?*
3. *В чем преимущества использования системы контроля версий?*
4. *Перечислите виды СКВ.*
5. *Дайте краткую характеристику каждого вида СКВ.*
6. *Что такое Git?*
7. *Опишите подход Git’а к работе со своими данными.*
8. *Охарактеризуйте три основных состояния, в которых могут находиться файлы.*
9. *Какие сервисы git вы знаете?*
10. *Как создать новый репозиторий?*
11. *Как опубликовать репозиторий Git?*

**Практическая работа №2**

**Тема**

Редактирование файла readme.md. Размещение файлов в репозиторий.

**Цель работы**

Получить начальный опыт размещения файлов в репозитории

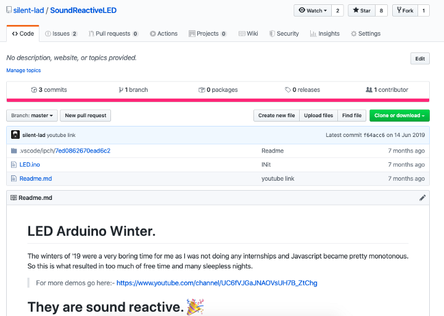
**Задание**

1. Изучите указанный теоретический материал.
2. Отредактируйте файл-шаблон readme.md.
3. Разместите отредактированный файл в репозиторий.
4. *Оформите отчет.*

**Теоретический материал**

README (буквально означает «прочти меня») — это первый файл, который нужно читать, получив доступ к проекту на Github или любой Git-хостинговой площадке. Этот файл в первую очередь и предлагается вниманию пользователя, когда он открывает здесь репозиторий того или иного проекта. Такой файл содержит много полезной информации, так что его вполне можно рассматривать как справочное руководство по проекту.

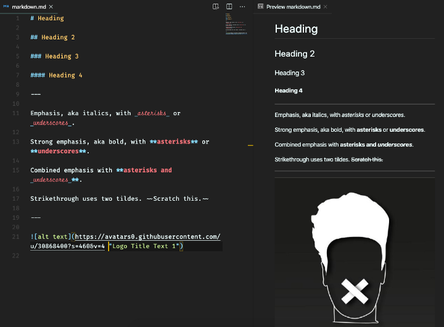
Посмотрите, где у нас здесь файл Readme:



Файл *Readme.md* находится в корневой папке репозитория и автоматически отображается в каталоге проекта на github.

Расширение .md — это сокращение от слова markdown. Это язык разметки для форматирования текста. Его используют (как и язык разметки HTML) для нормального отображения документов.

Вот как выглядит файл разметки на github (здесь использован VSCode, который одновременно показывает нам файлы разметки и в режиме предварительного просмотра):



**Git: основные команды**

1. **Git init**

Чтобы создать в текущей директории поддиректорию .git, где будут храниться все файлы репозитория, выполните git init. Если поддиректория уже существует, при повторной аналогичной команде будет выведено сообщение Reinitialized existing Git repository in.

1. **Git**add

Вам необходимо добавить в проект файлы с определенным расширением. Можно добавить все файлы по одному, но лучше использовать \*.< имя\_расширения>, чтобы включить все файлы с этим расширением:

git add \*.py

Если вы хотите добавить файлы с определенным расширением и следом указать имя каталога, то можно выполнить следующую команду. Она добавит все **.py** файлы из подкаталогов **models/directory**:

git add models/\\*.py

1. **Git**clean

Допустим, вы создали несколько новых файлов или папок в ветке Git, а через время оказалось, что эти файлы больше не нужны. В этом случае нужно очистить свое рабочее дерево от лишних файлов (которые были добавлены с помощью **git add**) следующей командой:

git clean -df

Чтобы увидеть, какие untracked-файлы будут удалены, используйте команду:

git clean -dn

1. **Git**commit

Вы закоммитили изменения, позже поняли, что допустили ошибку, или просто необходимо сделать описание более понятным. Тогда эта команда – то, что нужно:

git commit --amend -m "сообщение"

Если вы добавили новые файлы или исправили ошибку, но не хотите добавлять еще одно commit-сообщение, используйте команду с флагом **--no-edit**:

git commit --amend --no-edit

Вы запушили свой код в удаленный репозиторий, а затем поняли, что нужно изменить commit-сообщение. Для этого после внесения изменений сделайте принудительный **push**. Используя имя удаленного репозитория в качестве источника, выполните:

git push origin <имя ветки> -f

1. **Git  config**

После настройки удаленного репозитория его URL-адрес нужно добавить в локальный файл git config, а также создать вышестоящую ветку для локальных веток. Такую возможность предоставляет команда git remote.

git remote add <remote\_name> <remote\_repo\_url>

Эта команда привяжет удаленный репозиторий по адресу  к ссылке в вашем локальном репозитории . После привязки удаленного репозитория в него можно будет отправлять локальные ветки с помощью команды push.

git push -u <remote\_name> <local\_branch\_name>

Эта команда поместит ветку локального репозитория с именем < local\_branc\_name > в удаленный репозиторий < remote\_name >.

Дополнительную информацию о команде git remote см. на [странице удаленной работы в Git](https://www.atlassian.com/git/tutorials/syncing#git-remote).

Помимо конфигурирования URL-адреса удаленного репозитория, вам может потребоваться установить глобальные параметры Git, например имя пользователя или электронный адрес. Команда git config позволяет настроить инсталляцию Git (или отдельный репозиторий) из командной строки. С помощью этой команды можно установить любые настройки: от информации о пользователе до его предпочтений и характеристик репозитория. Ниже перечислены распространенные варианты конфигурации.

Git хранит варианты конфигурации в трех различных файлах, позволяющих ограничивать область видимости на уровне отдельных репозиториев (локальный), пользователя (глобальный) или всей системы (системный):

* Локальный: /.git/config — настройки на уровне репозитория.
* Глобальный: /.gitconfig — настройки на уровне пользователя. Здесь хранятся настройки с флагом --global.
* Системный: $(prefix)/etc/gitconfig — настройки на уровне всей системы.

Укажите имя автора, которое будет использоваться для всех коммитов в текущем репозитории. Обычно для настройки параметров конфигурации для текущего пользователя используется флаг --global.

git config --global user.name <name>

Эта команда задает имя автора, которое будет использоваться для всех коммитов, выполненных текущим пользователем.

Добавление аргумента --local или выполнение команды без параметра уровня конфигурации приведет к установке значения user.name для текущего локального репозитория.

git config --local user.email <email>

Эта команда задает адрес электронной почты автора, который будет использоваться для всех коммитов, выполненных текущим пользователем.

1. **Git**push

Если git pull предназначен для мержа изменений в локальный репозиторий из удаленного, то git push действует с точностью до наоборот: локальные изменения пушатся в удаленный репозиторий.

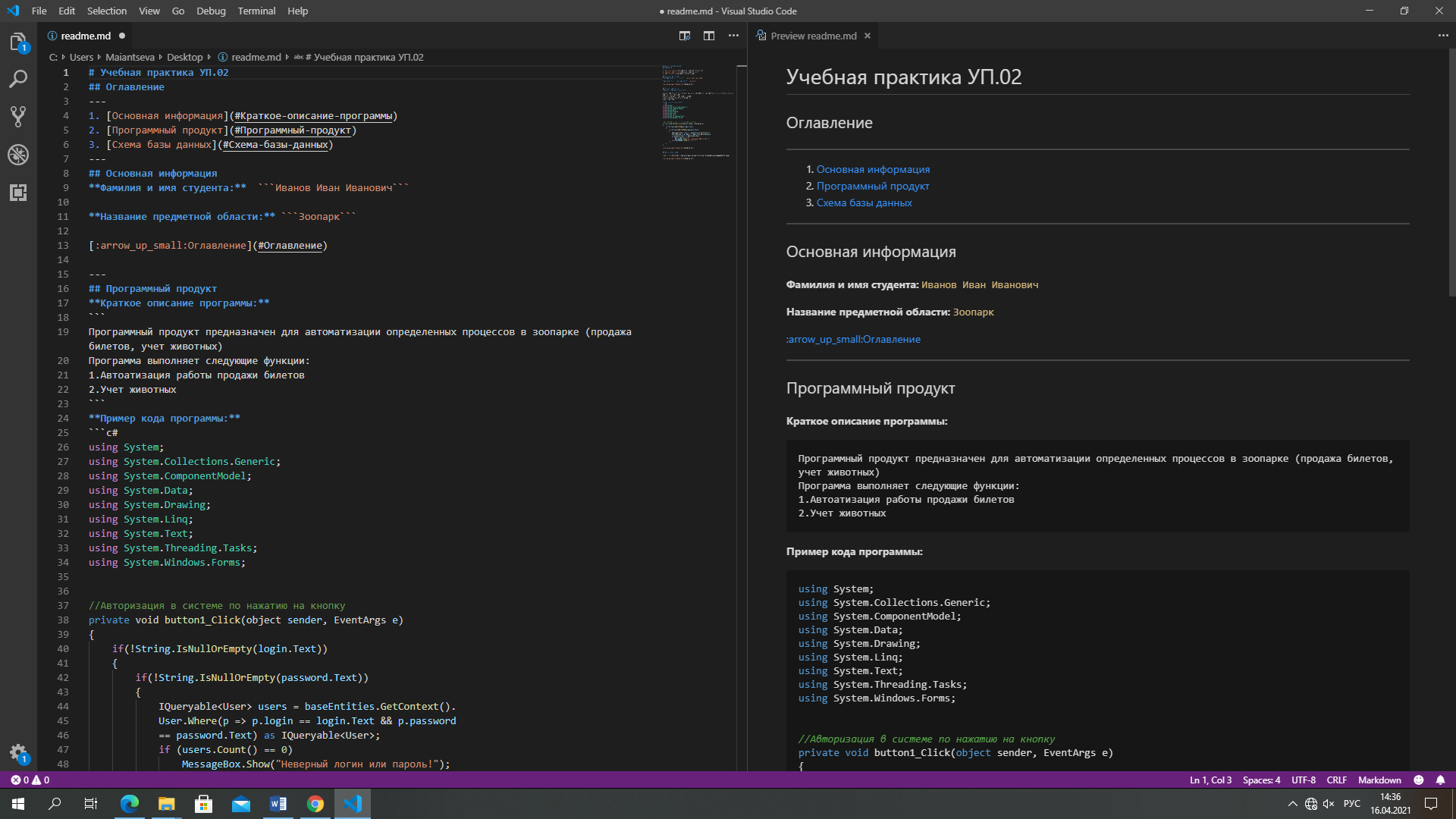
Также можно использовать:

git push origin master

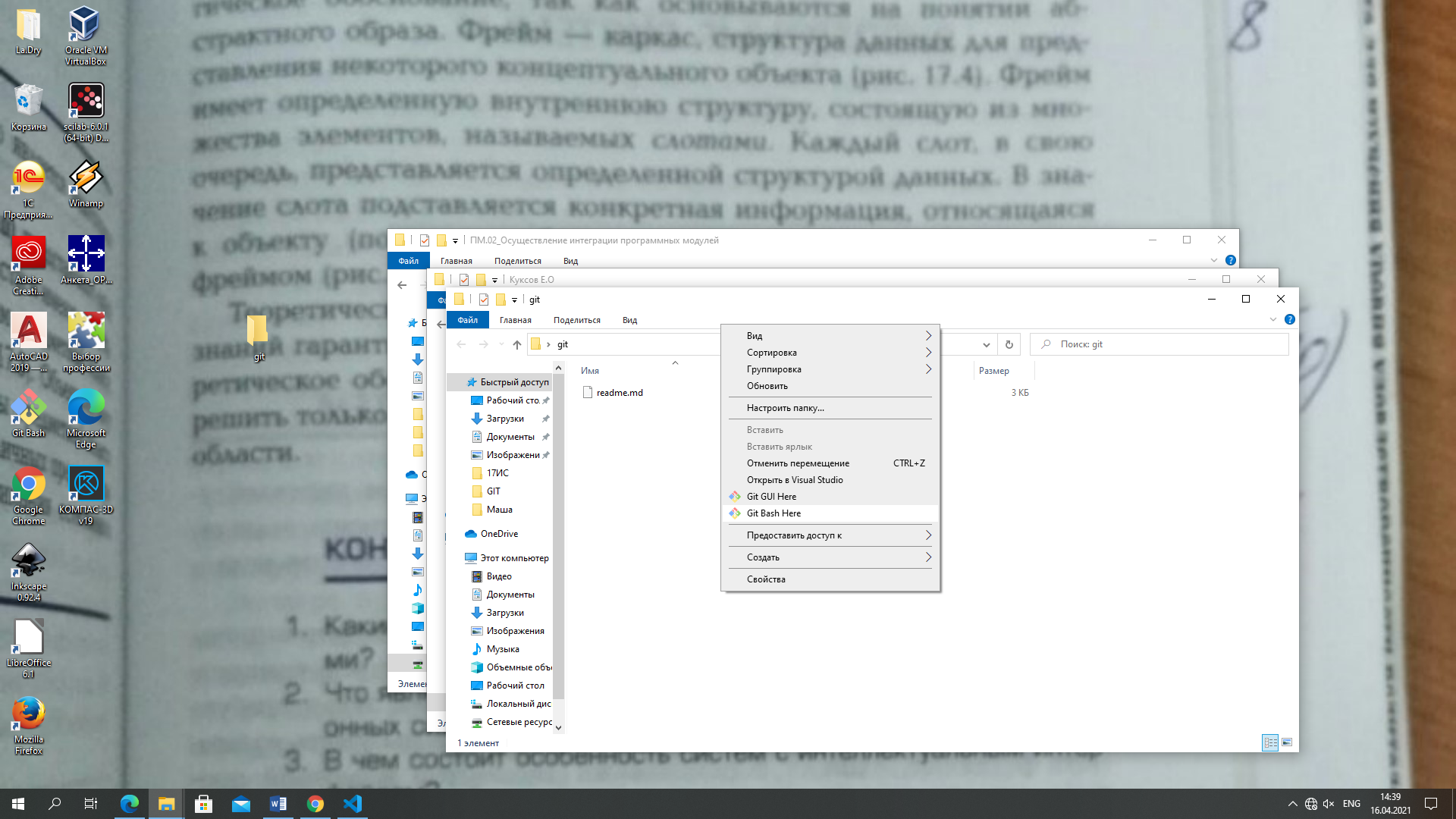
Что означает коммит из локальной ветки master на удаленный репозиторий origin.

**Пояснение к работе**

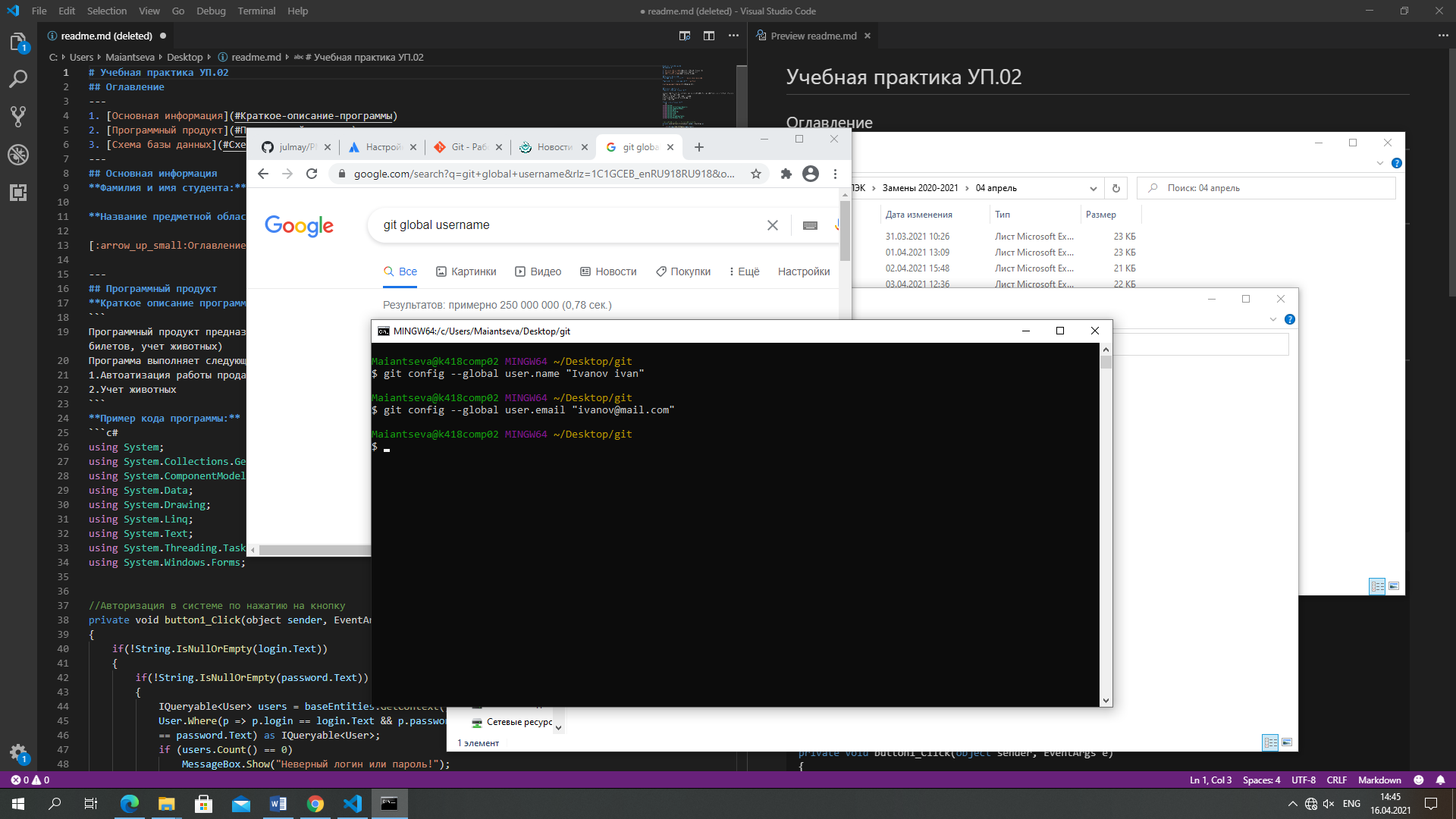
1. Отредактируйте файл readme.md согласно шаблону (Удобнее это сделать в редакторе Visual Codeтак как в нем имеется возможность предварительного просмотра)



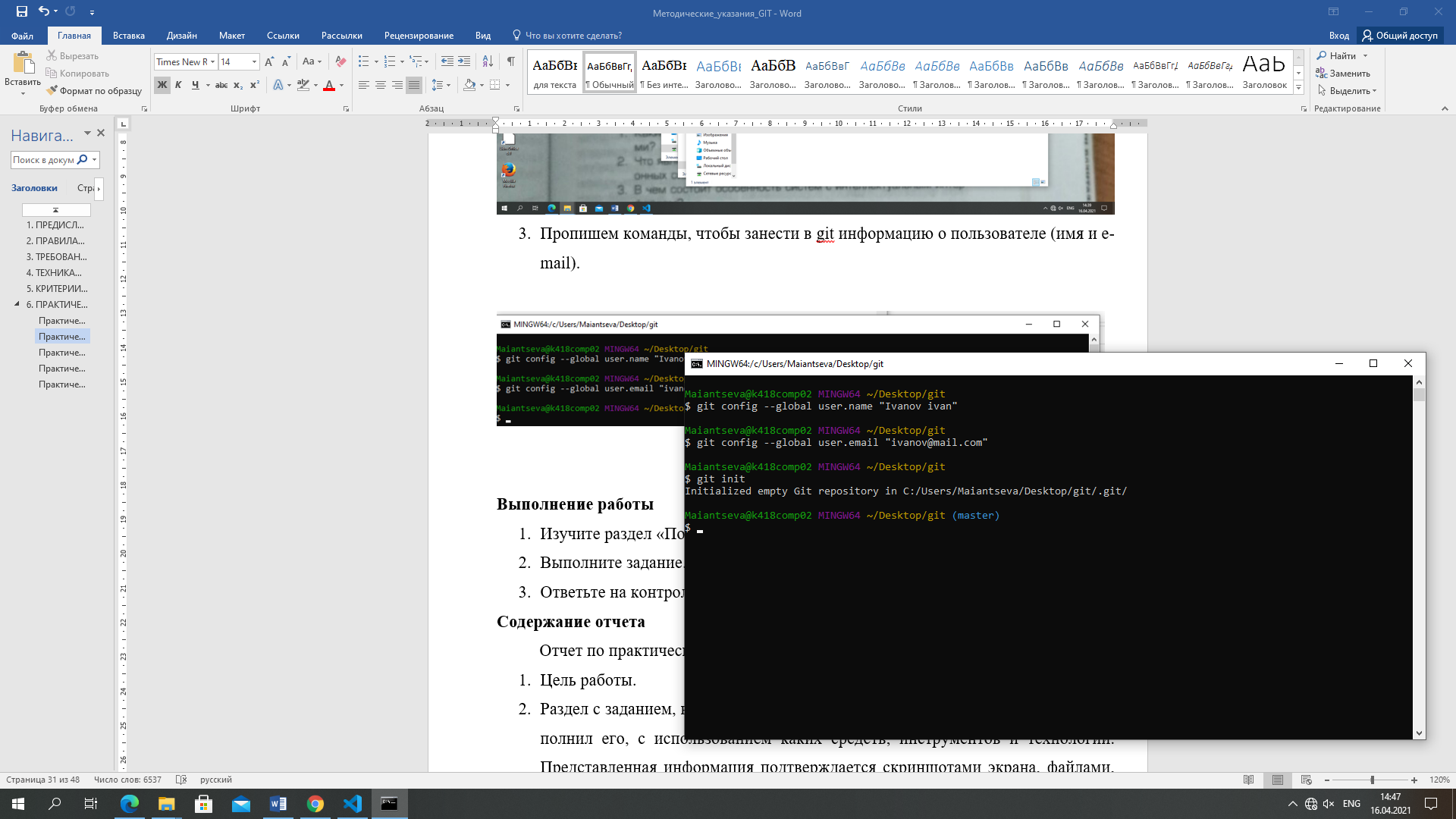
1. Запустим программу git в режиме консоли (для этого перейдем в каталог с файлом readme.md, правой кнопкой мыши нажмем в свободное пространство и выберем Git Bash Here).



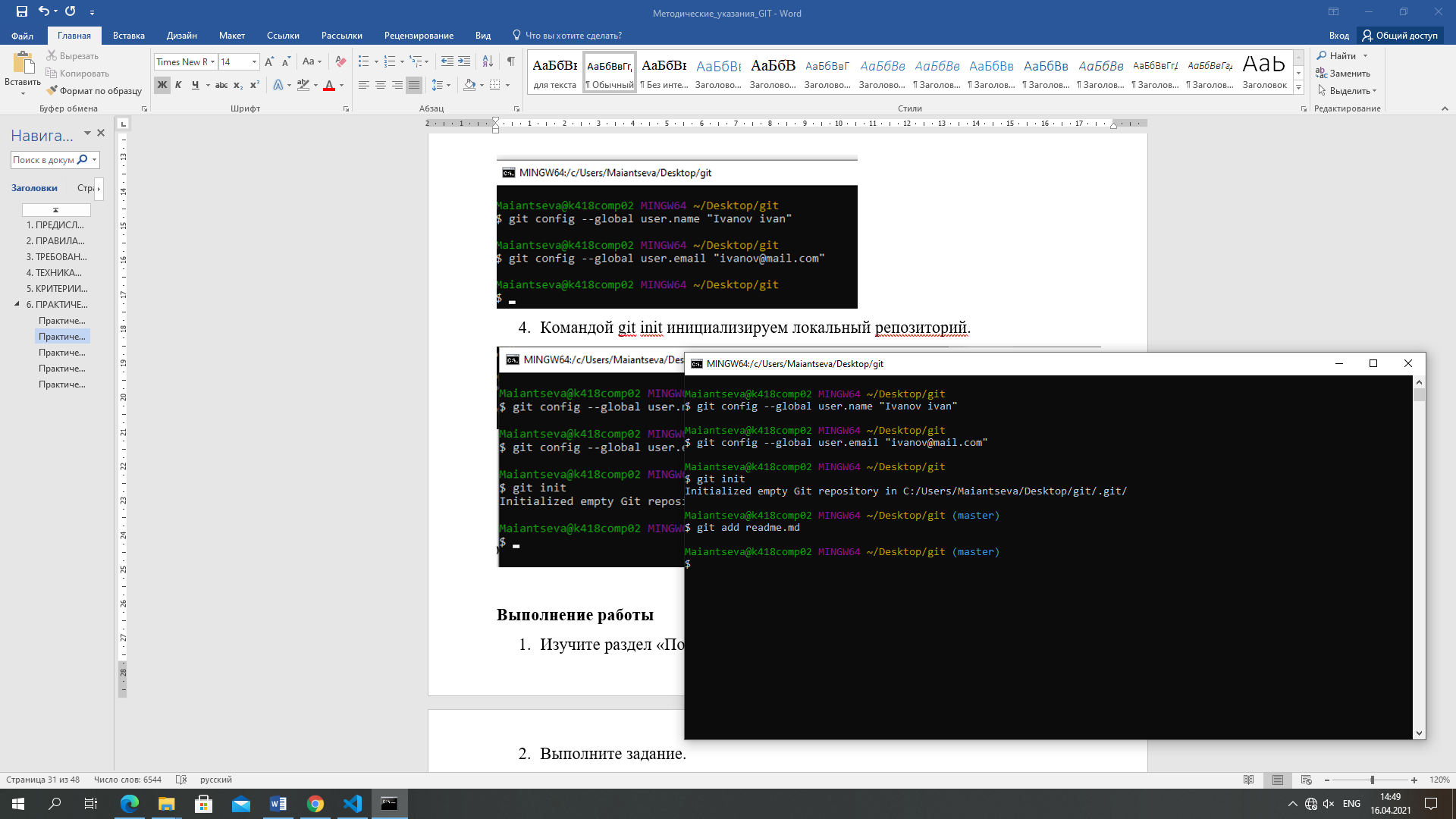
1. Пропишем команды, чтобы занести в git информацию о пользователе (имя и e-mail).



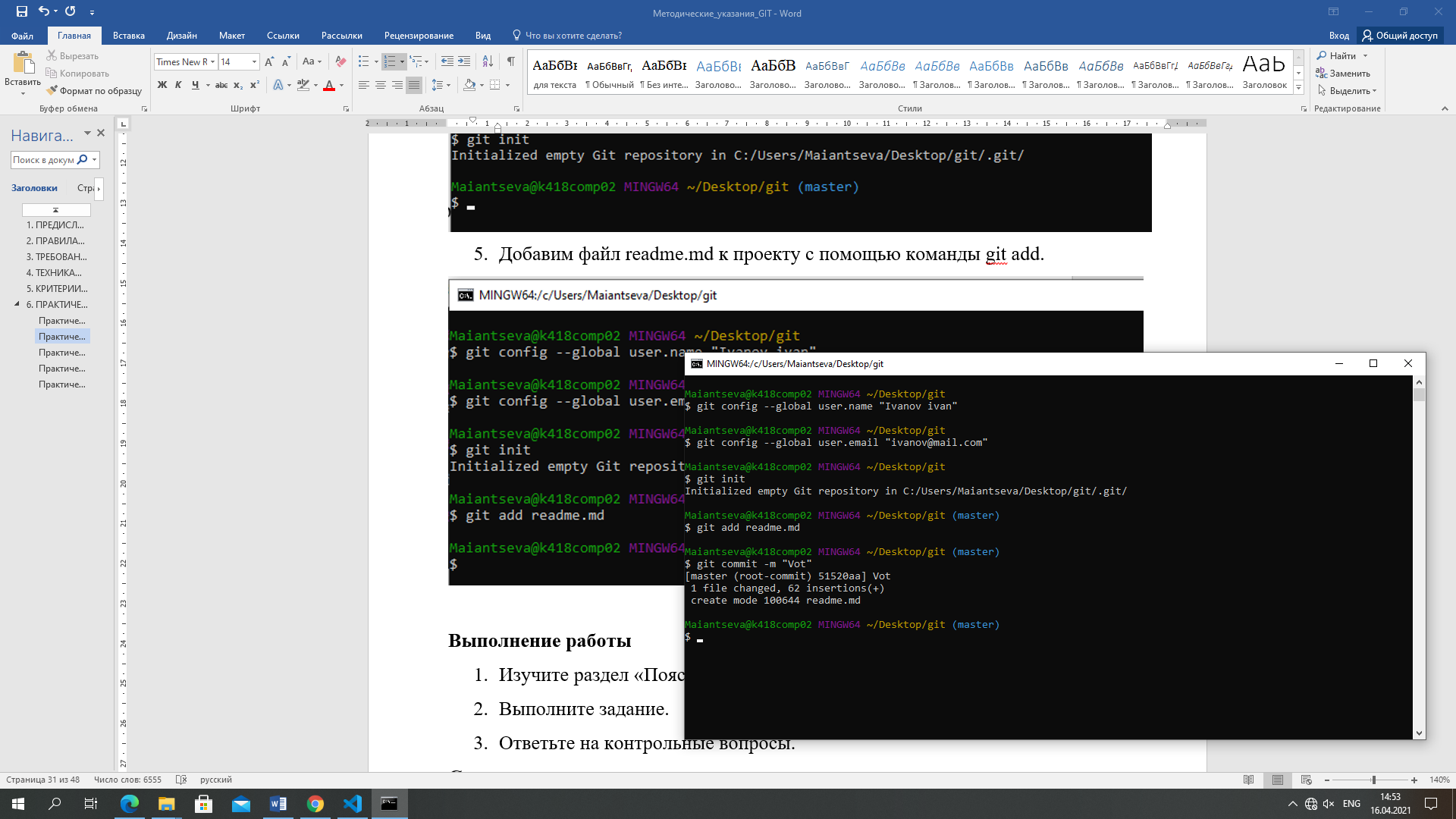
1. Командой git init инициализируем локальный репозиторий.



1. Добавим файл readme.md к проекту с помощью команды git add.

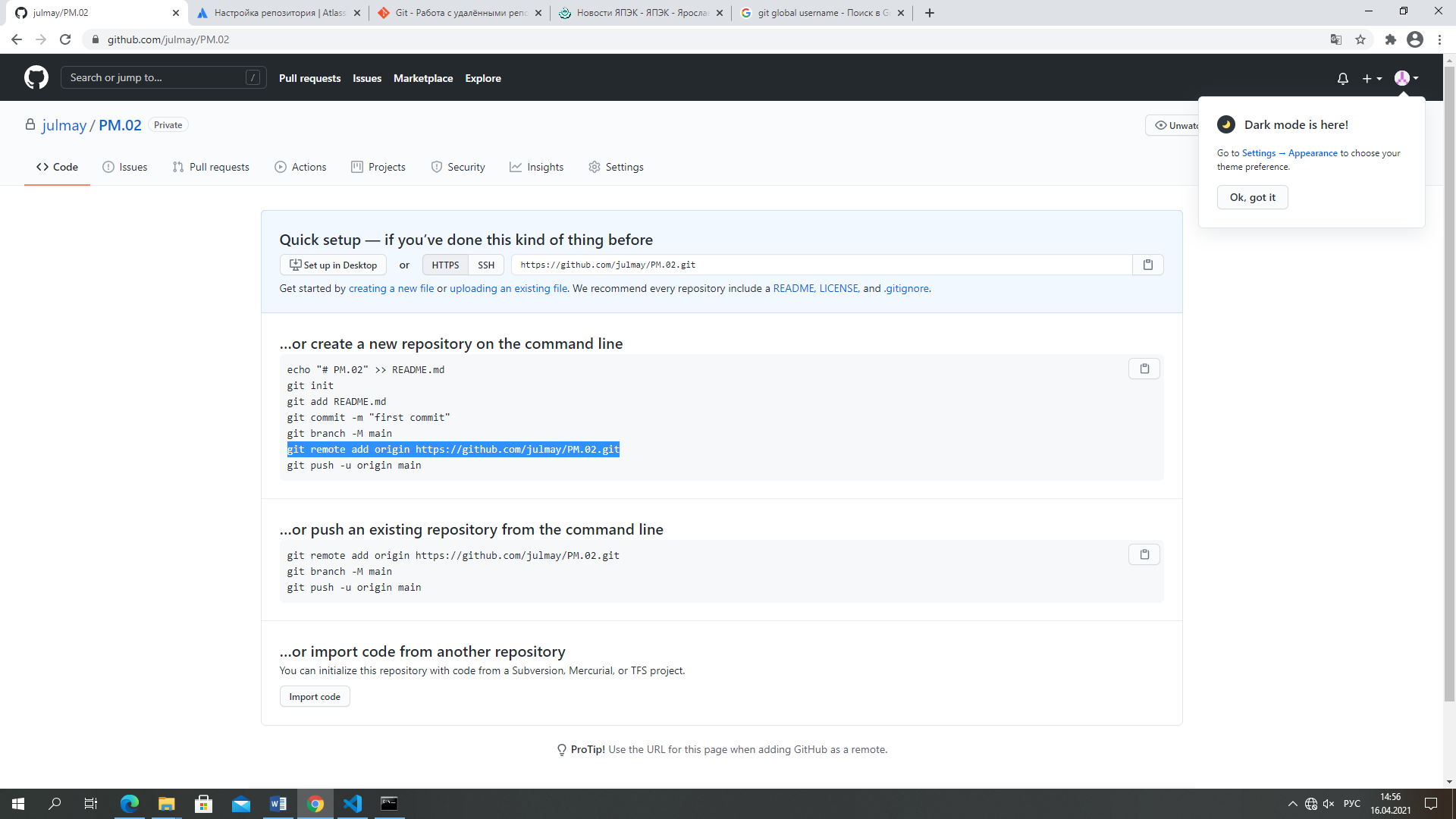


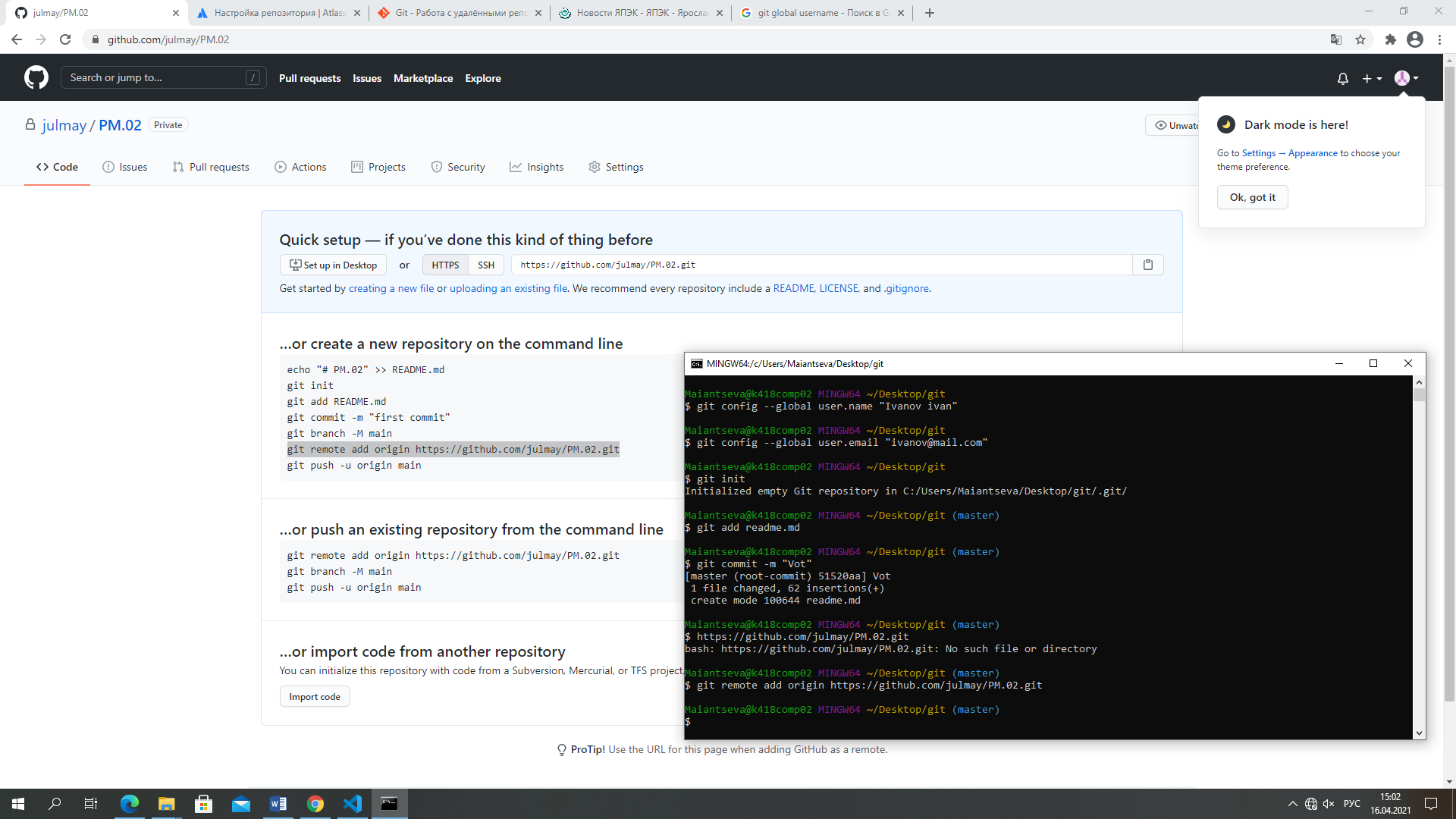
1. Создадим коммит с помощью команды git commit. Параметр –m добавляет комментарий к коммиту.



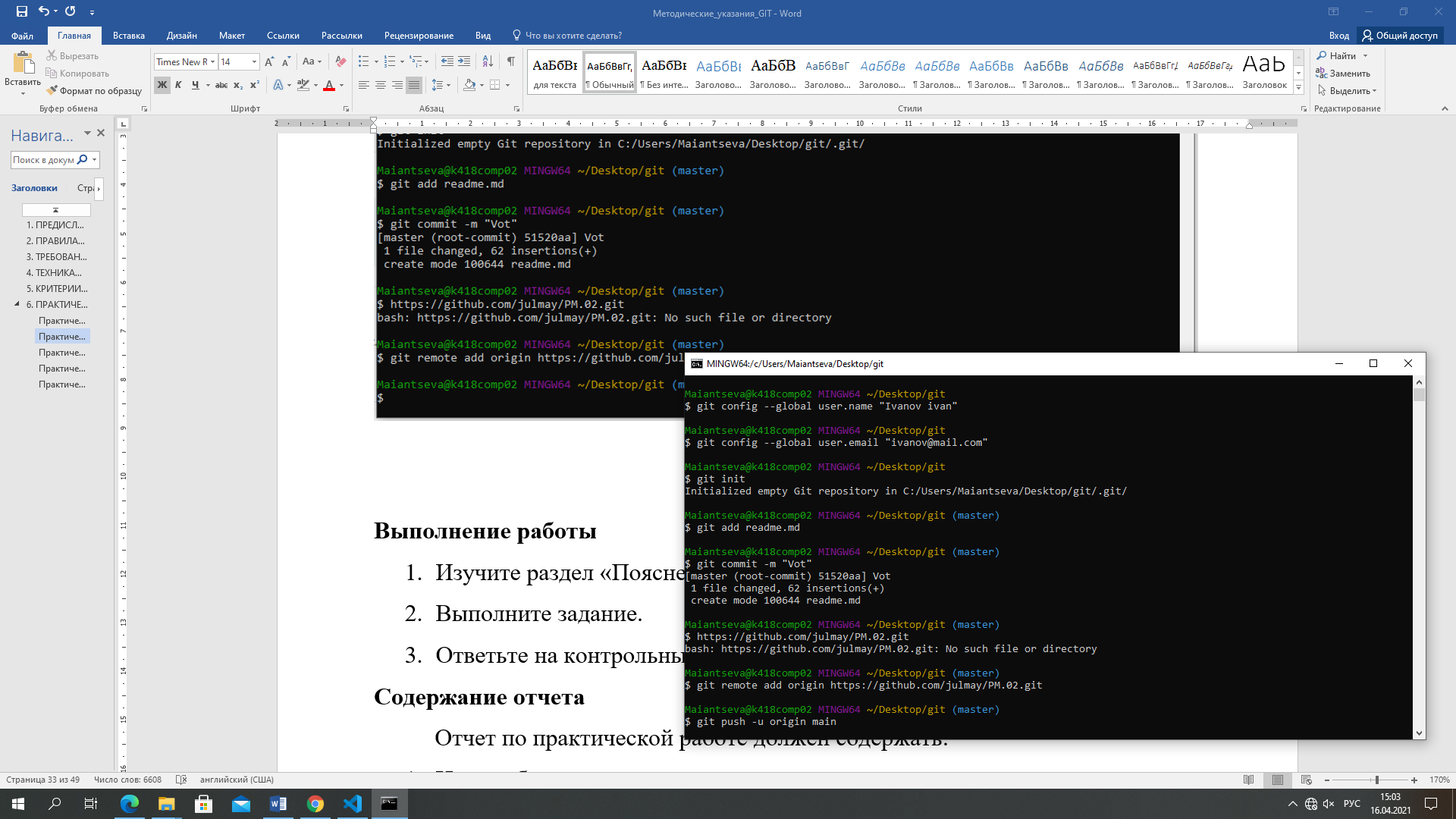
1. Подключимся к удаленному репозиторию с помощью команды git remote add origin <https://github.com/...git> (где origin - название репозитория (можно указать любое), а <https://github.com/...git> – ссылка на ваш репозиторий).

*Если репозиторий только что создан, то можно скопировать строку с командой в github.*



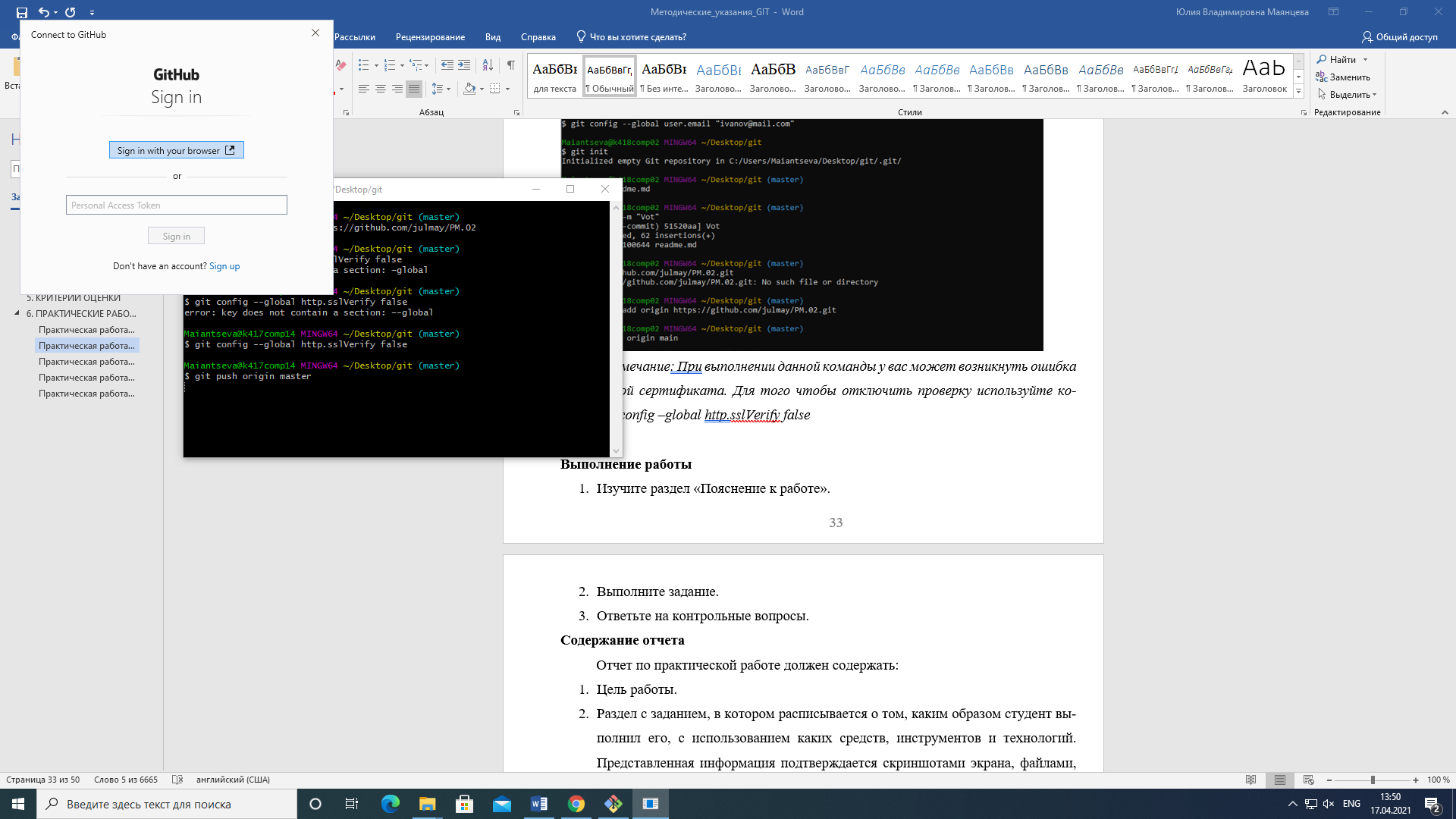


1. С помощью команды git push отразим изменения в локальном репозитории на удаленном. После параметра –u указывается название удаленного репозитория, а затем название ветки локального репозитория (т.к. мы не создавали ветки (про это мы поговорим позже), то выгружать мы будем с главной ветки, т.е. main).



*Примечание: При выполнении данной команды у вас может возникнуть ошибка с проверкой сертификата. Для того чтобы отключить проверку используйте команду git config --global http.sslVerify false*

1. Появится окно авторизации github. Нажимаем на кнопку «Sign in ..»



**Выполнение работы**

1. Изучите раздел «Пояснение к работе».
2. Выполните задание.
3. Ответьте на контрольные вопросы.

**Содержание отчета**

Отчет по практической работе должен содержать:

1. Цель работы.
2. Раздел с заданием, в котором расписывается о том, каким образом студент выполнил его, с использованием каких средств, инструментов и технологий. Представленная информация подтверждается скриншотами экрана, файлами, полученными в ходе выполнения практической работы.
3. Ответы на контрольные вопросы.

***Контрольные вопросы***

1. *Как создать новую ветвь?*
2. *Как просмотреть историю изменений?*
3. *Как сохранять и синхронизировать изменения?*
4. *Как выполнить слияние ветвей?*

**Практическая работа №3**

**Тема**

Ветки.

**Цель работы**

Получить практический опыт в создании веток в git.

**Задание**

**Ветвление в Git - О ветвлении в двух словах**

Почти каждая система контроля версий (СКВ) в какой-то форме поддерживает ветвление. Используя ветвление, Вы отклоняетесь от основной линии разработки и продолжаете работу независимо от неё, не вмешиваясь в основную линию. Во многих СКВ создание веток — это очень затратный процесс, часто требующий создания новой копии каталога с исходным кодом, что может занять много времени для большого проекта.

Некоторые люди, говоря о модели ветвления Git, называют ее «киллер-фича», что выгодно выделяет Git на фоне остальных СКВ. Что в ней такого особенного? Ветвление Git очень легковесно: операция создания ветки выполняется почти мгновенно, переключение между ветками туда-сюда, обычно, также быстро. В отличие от многих других СКВ, Git поощряет процесс работы, при котором ветвление и слияние выполняется часто, даже по несколько раз в день. Понимание и владение этой функциональностью дает вам уникальный и мощный инструмент, который может полностью изменить привычный процесс разработки.

**О ветвлении в двух словах**

Для точного понимания механизма ветвлений, необходимо вернуться назад и изучить то, как Git хранит данные.

Как вы можете помнить из [Что такое Git?](https://git-scm.com/book/ru/v2/ch00/what_is_git_section), Git не хранит данные в виде последовательности изменений, он использует набор снимков (snapshot).

Когда вы делаете коммит, Git сохраняет его в виде объекта, который содержит указатель на снимок (snapshot) подготовленных данных. Этот объект так же содержит имя автора и email, сообщение и указатель на коммит или коммиты непосредственно предшествующие данному (его родителей): отсутствие родителя для первоначального коммита, один родитель для обычного коммита, и несколько родителей для результатов слияния двух и более веток.

Предположим, у вас есть каталог с тремя файлами и вы добавляете их все в индекс и создаёте коммит. Во время индексации вычисляется контрольная сумма каждого файла (SHA-1 как мы узнали из [Что такое Git?](https://git-scm.com/book/ru/v2/ch00/what_is_git_section)), затем каждый файл сохраняется в репозиторий (Git называет такой файл блоб — большой бинарный объект), а контрольная сумма попадёт в индекс:

$ git add README test.rb LICENSE

$ git commit -m 'Initial commit'

Когда вы создаёте коммит командой git commit, Git вычисляет контрольные суммы каждого подкаталога (в нашем случае, только основной каталог проекта) и сохраняет его в репозитории как объект дерева каталогов. Затем Git создаёт объект коммита с метаданными и указателем на основное дерево проекта для возможности воссоздать этот снимок в случае необходимости.

Ваш репозиторий Git теперь хранит пять объектов: три блоб объекта (по одному на каждый файл), объект дерева каталогов, содержащий список файлов и соответствующих им блобов, а так же объект коммита, содержащий метаданные и указатель на объект дерева каталогов.

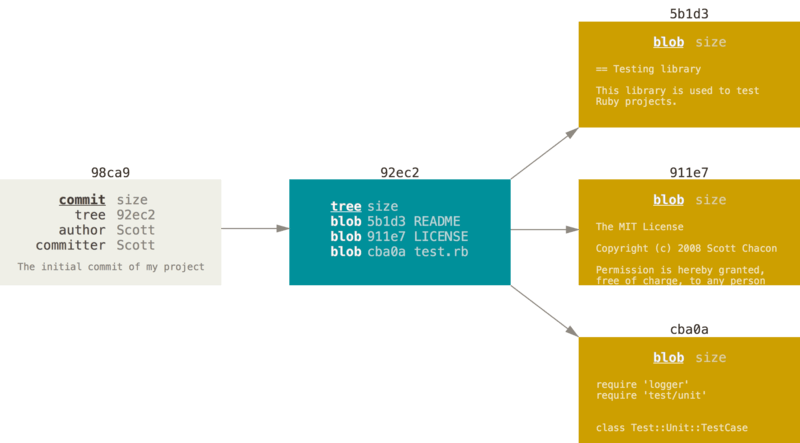


Рисунок. Коммит и его дерево

Если вы сделаете изменения и создадите ещё один коммит, то он будет содержать указатель на предыдущий коммит.

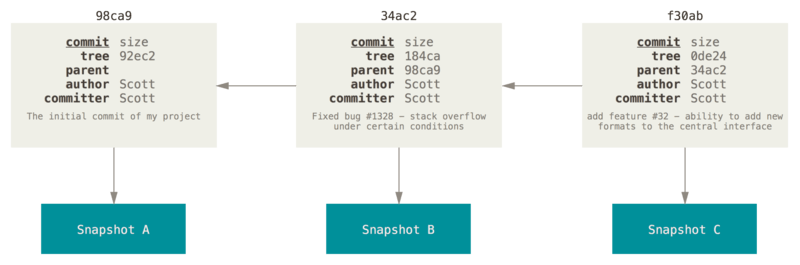


Рисунок. Коммит и его родители

Ветка в Git — это простой перемещаемый указатель на один из таких коммитов. По умолчанию, имя основной ветки в Git — master. Как только вы начнёте создавать коммиты, ветка master будет всегда указывать на последний коммит. Каждый раз при создании коммита указатель ветки master будет передвигаться на следующий коммит автоматически.

|  |  |
| --- | --- |
| **Примечание** | Ветка «master» в Git — это не какая-то особенная ветка. Она точно такая же, как и все остальные ветки. Она существует почти во всех репозиториях только лишь потому, что её создаёт команда git init, а большинство людей не меняют её название. |

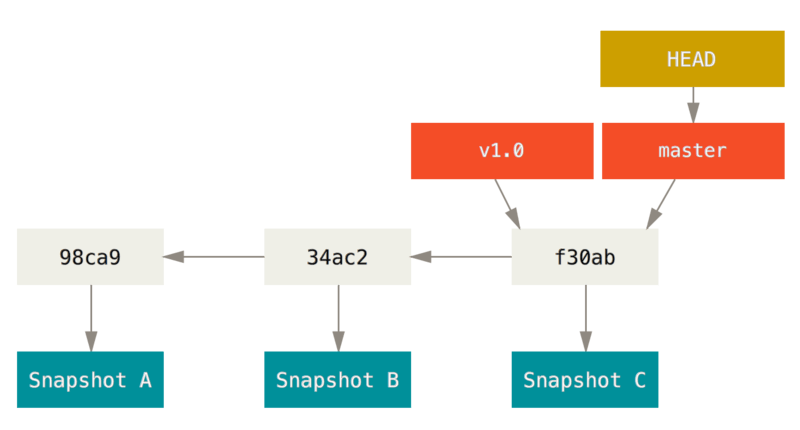


Рисунок. Ветка и история коммитов

**Создание новой ветки**

Что же на самом деле происходит при создании ветки? Всего лишь создаётся новый указатель для дальнейшего перемещения. Допустим вы хотите создать новую ветку с именем testing. Вы можете это сделать командой git branch :

$ git branch testing

В результате создаётся новый указатель на текущий коммит.

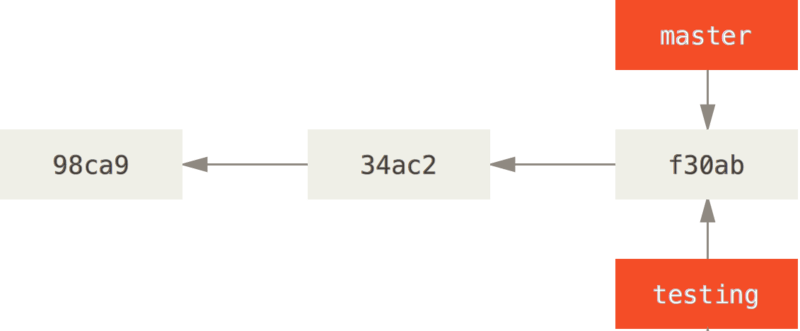


Рисунок 12. Две ветки указывают на одну и ту же последовательность коммитов

Как Git определяет, в какой ветке вы находитесь? Он хранит специальный указатель HEAD. Имейте ввиду, что в Git концепция HEAD значительно отличается от других систем контроля версий, которые вы могли использовать раньше (Subversion или CVS). В Git — это указатель на текущую локальную ветку. В нашем случае мы все еще находимся в ветке master. Команда git branch только создаёт новую ветку, но не переключает на неё.

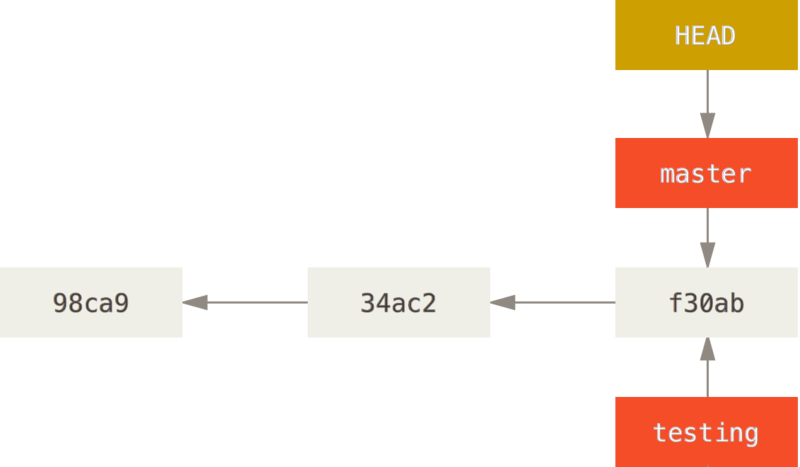


Рисунок. HEAD указывает на ветку

Вы можете легко это увидеть при помощи простой команды git log, которая покажет вам куда указывают указатели веток. Эта опция называется --decorate.

$ git log --oneline --decorate

f30ab (HEAD -> master, testing) Add feature #32 - ability to add new formats to the central interface

34ac2 Fix bug #1328 - stack overflow under certain conditions

98ca9 Initial commit

Здесь можно увидеть указывающие на коммит f30ab ветки: master и testing.

Переключение веток

Для переключения на существующую ветку выполните команду git checkout. Давайте переключимся на ветку testing:

$ git checkout testing

В результате указатель HEAD переместится на ветку testing.

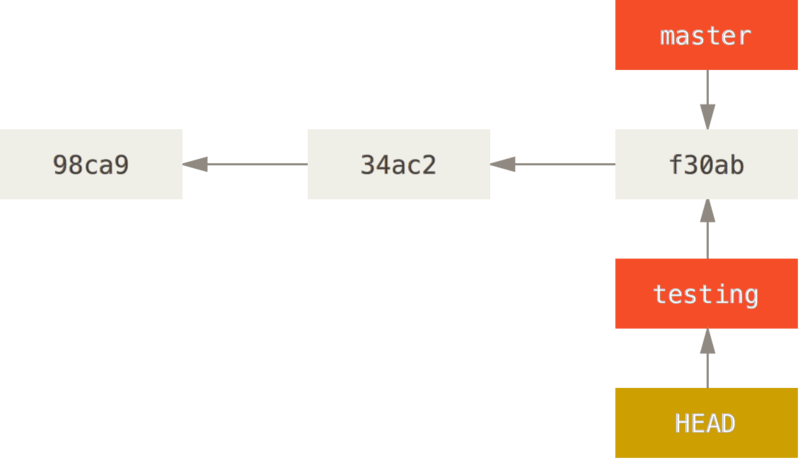


Рисунок. HEAD указывает на текущую ветку

Какой в этом смысл? Давайте сделаем ещё один коммит:

$ vim test.rb

$ git commit -a -m 'made a change'

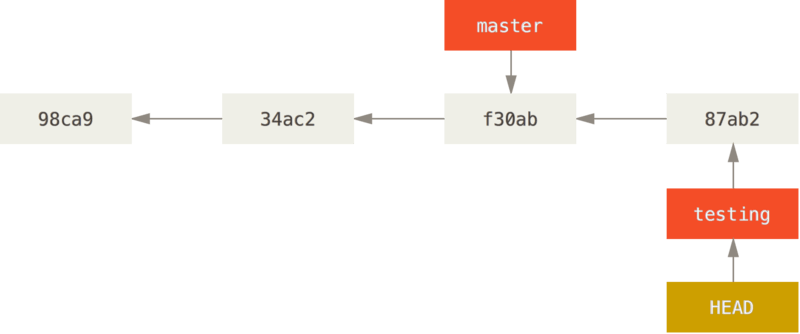


Рисунок. Указатель на ветку HEAD переместился вперёд после коммита

Интересная ситуация: указатель на ветку testing переместился вперёд, а master указывает на тот же коммит, где вы были до переключения веток командой git checkout. Давайте переключимся назад на ветку master:

$ git checkout master

|  |  |
| --- | --- |
| **Примечание** | **git log не показывает все ветки по умолчанию**  Если выполнить команду git log прямо сейчас, то в её выводе только что созданная ветка «testing» фигурировать не будет.  Ветка никуда не исчезла; просто Git не знает, что именно она вас интересует, и выводит наиболее полезную по его мнению информацию. Другими словами, по умолчанию git log отобразит историю коммитов только для текущей ветки.  Для просмотра истории коммитов другой ветки необходимо явно указать её имя: git log testing Чтобы посмотреть историю по всем веткам — выполните команду с дополнительным флагом: git log --all. |

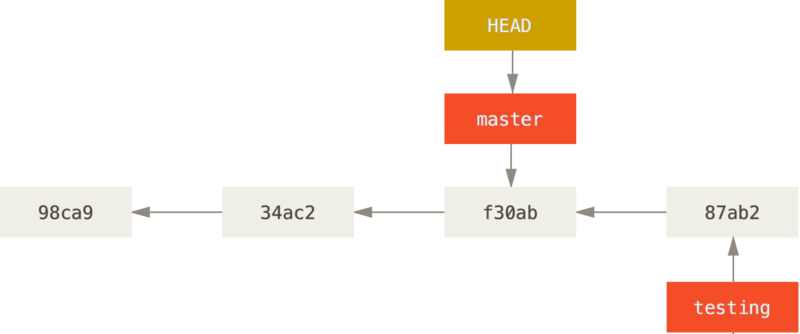


Рисунок. HEAD перемещается когда вы делаете checkout

Эта команда сделала две вещи: переместила указатель HEAD назад на ветку master и вернула файлы в рабочем каталоге в то состояние, на снимок которого указывает master. Это также означает, что все вносимые с этого момента изменения будут относиться к старой версии проекта. Другими словами, вы откатили все изменения ветки testing и можете продолжать в другом направлении.

|  |  |
| --- | --- |
| **Примечание** | **Переключение веток меняет файлы в рабочем каталоге**  Важно запомнить, что при переключении веток в Git происходит изменение файлов в рабочем каталоге. Если вы переключаетесь на старую ветку, то рабочий каталог будет выглядеть так же, как выглядел на момент последнего коммита в ту ветку. Если Git по каким-то причинам не может этого сделать — он не позволит вам переключиться вообще. |

Давайте сделаем еще несколько изменений и создадим очередной коммит:

$ vim test.rb

$ git commit -a -m 'made other changes'

Теперь история вашего проекта разошлась. Вы создали ветку и переключились на нее, поработали, а затем вернулись в основную ветку и поработали в ней. Эти изменения изолированы друг от друга: вы можете свободно переключаться туда и обратно, а когда понадобится — объединить их. И все это делается простыми командами: branch, checkout и commit.

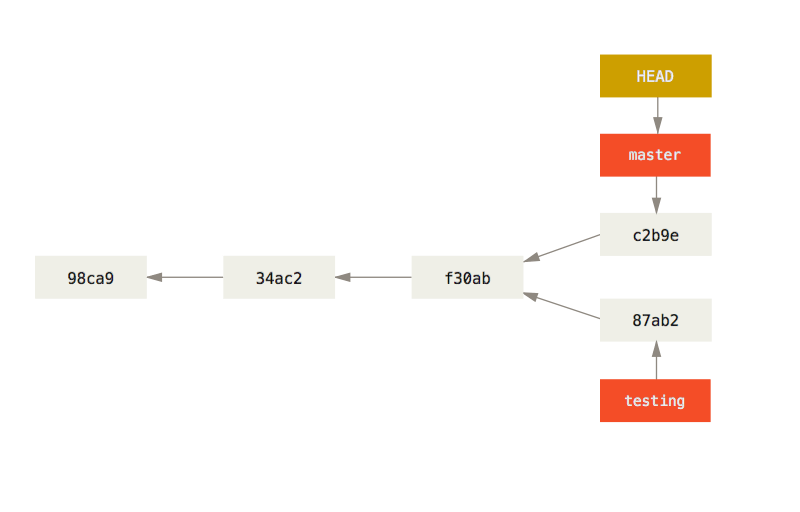


Рисунок. Разветвлённая история

Все описанные действия можно визуализировать с помощью команды git log. Для отображения истории коммитов, текущего положения указателей веток и истории ветвления выполните команду git log --oneline --decorate --graph --all.

$ git log --oneline --decorate --graph --all

\* c2b9e (HEAD, master) Made other changes

| \* 87ab2 (testing) Made a change

|/

\* f30ab Add feature #32 - ability to add new formats to the central interface

\* 34ac2 Fix bug #1328 - stack overflow under certain conditions

\* 98ca9 initial commit of my project

Ветка в Git — это простой файл, содержащий 40 символов контрольной суммы SHA-1 коммита, на который она указывает; поэтому операции с ветками являются дешёвыми с точки зрения потребления ресурсов или времени. Создание новой ветки в Git происходит так же быстро и просто как запись 41 байта в файл (40 знаков и перевод строки).

Это принципиально отличает процесс ветвления в Git от более старых систем контроля версий, где все файлы проекта копируются в другой подкаталог. В зависимости от размера проекта, операции ветвления в таких системах могут занимать секунды или даже минуты, когда в Git эти операции мгновенны. Поскольку при коммите мы сохраняем указатель на родительский коммит, то поиск подходящей базы для слияния веток делается автоматически и, в большинстве случаев, очень прост. Эти возможности побуждают разработчиков чаще создавать и использовать ветки.

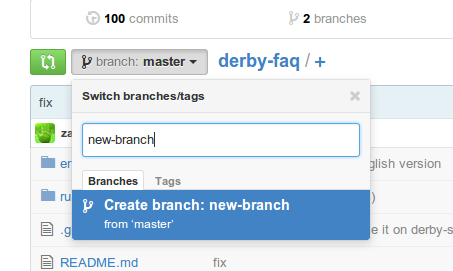
Давайте посмотрим, почему и вам имеет смысл делать так же.

|  |  |
| --- | --- |
| **Примечание** | **Одновременное создание новой ветки и переключение на неё**  Как правило, при создании новой ветки вы хотите сразу на неё переключиться — это можно сделать используя команду git checkout -b <newbranchname>. |
| **Примечание** | Начиная с Git версии 2.23, вы можете использовать git switch вместо git checkout, чтобы:   * Переключиться на существующую ветку: git switch testing-branch. * Создать новую ветку и переключиться на нее: git switch -c new-branch. Флаг -c означает создание, но также можно использовать полный формат:` --create`. * Вернуться к предыдущей извлечённой ветке: git switch -. |

**Создание ветки в github**

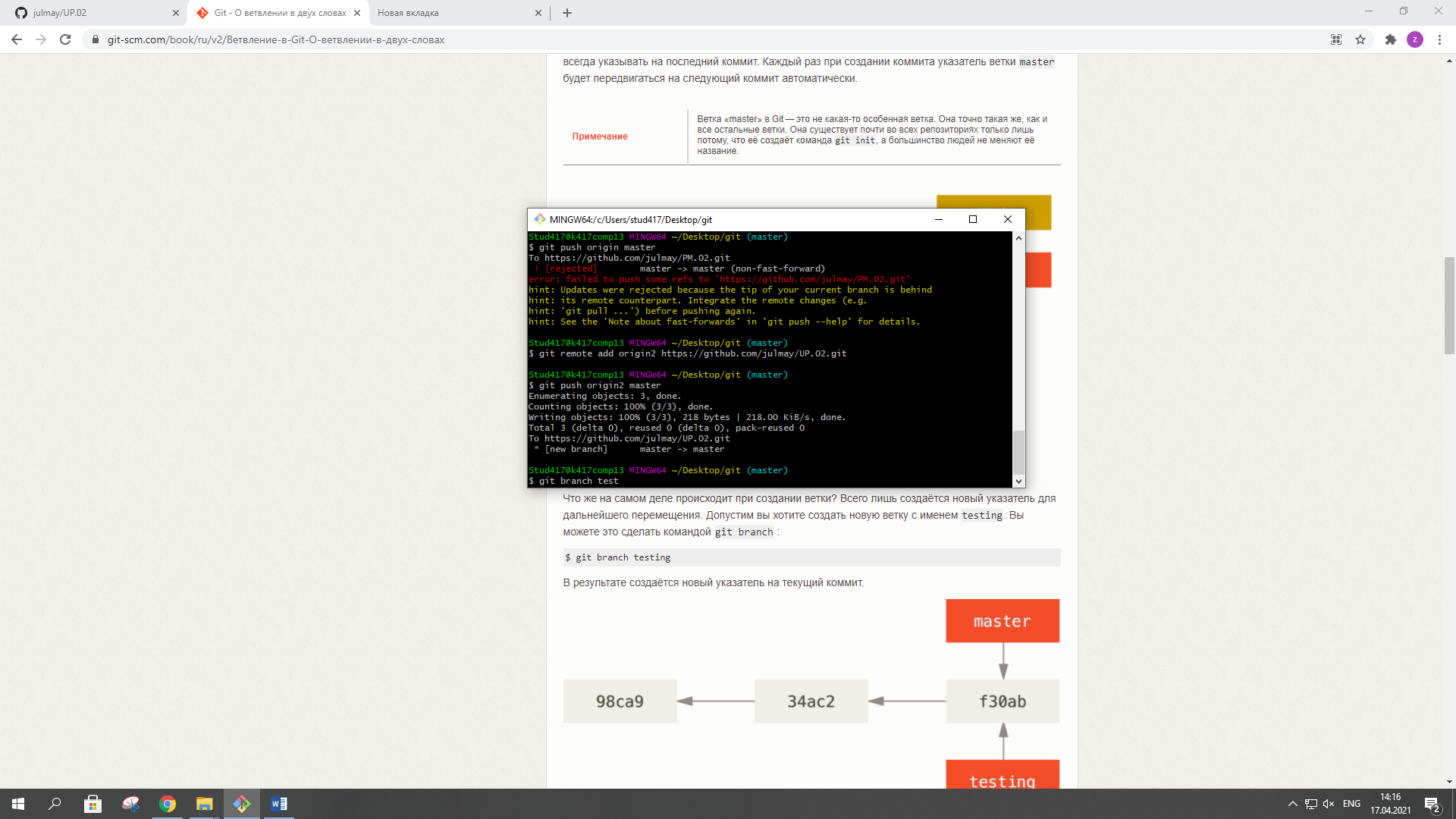
Считается признаком хорошего тона, если вы оформите свои правки в отдельной ветке, ведь хозяин оригинального репозитория может попросить вас что-то поменять/доработать перед слиянием.

Создать новую ветку (копированием из текущей) можно прямо в окошке смены ветки. Вводим имя — enter — готово.

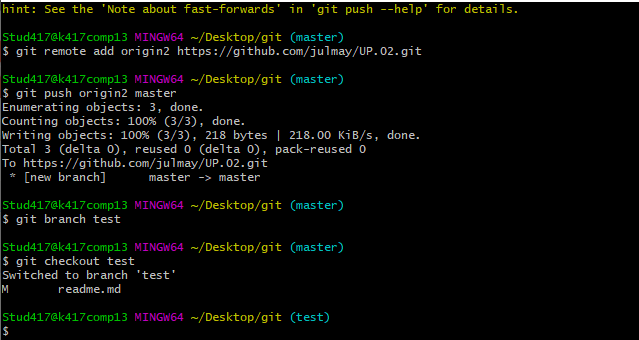


**Пояснение к работе**

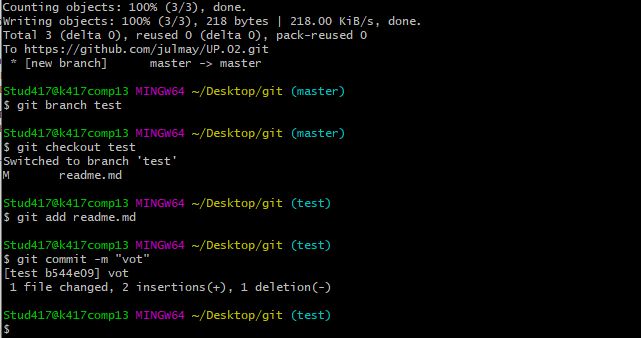
1. С помощью команды git branch, где test – название новой ветки.



1. Переключимся на созданную ветку с помощью команды git checkout, где test – название созданной ветки.



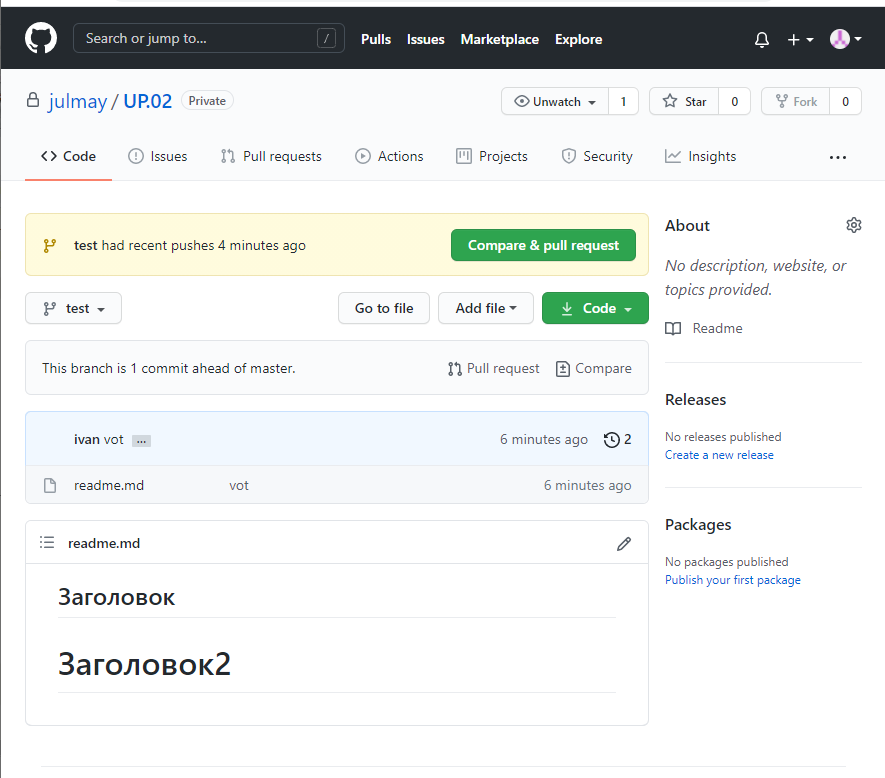
1. Для примера, добавим файл readme.md в проект и сделает коммит.



1. С помощью команды git push отразим имзенения на удаленном репозитории. Обратите внимание, что на уделнном репозитории была создана новая ветка.



1. Зайдите на github и убедитесь, что был создана новая ветка.



**Выполнение работы**

1. Изучите раздел «Пояснение к работе».
2. Создайте в своем репозитории ветку «Практическое задание».
3. Ответьте на контрольные вопросы.
4. Оформите отчет.

**Содержание отчета**

Отчет по практической работе должен содержать:

1. Цель работы.
2. Раздел с заданием, в котором расписывается о том, каким образом студент выполнил его, с использованием каких средств, инструментов и технологий. Представленная информация подтверждается скриншотами экрана, файлами, полученными в ходе выполнения практической работы.
3. Ответы на контрольные вопросы.

**Контрольные вопросы**

1. Как создать публичный репозиторий?
2. Как создать новую ветвь?
3. Как сохранить изменения в репозиторий?

**Практическая работа №4**

**Тема**

**Цель работы**

**Задание**

**Пояснение к работе**

**Выполнение работы**

1. Изучите раздел «Пояснение к работе».
2. Выполните задание.
3. Ответьте на контрольные вопросы.

**Содержание отчета**

Отчет по практической работе должен содержать:

1. Цель работы.
2. Раздел с заданием, в котором расписывается о том, каким образом студент выполнил его, с использованием каких средств, инструментов и технологий. Представленная информация подтверждается скриншотами экрана, файлами, полученными в ходе выполнения практической работы.
3. Ответы на контрольные вопросы.

**Контрольные вопросы**

1. Как просмотреть историю изменений?
2. Как сохранять и синхронизировать изменения?
3. Как выполнить слияние ветвей?

**Практическая работа №5**

**Тема**

**Цель работы**

**Задание**

1. Запустить Git Bash. Создать новый репозиторий.
2. Добавить в папку репозитория текстовый файл. Зафиксировать состояние репозитория (выполнить commit).
3. Внести изменения в файл. Зафиксировать новое состояние репозитория.
4. Создать новую ветку 1. Внести в нее изменения и зафиксировать их.
5. Переключиться на ветку мастера. Внести в нее изменения и зафиксировать их.
6. Продемонстрировать слияние веток.
7. Просмотреть дерево изменений веток (историю).
8. Продемонстрировать откат изменений в ветке 1.

**Пояснение к работе**

**Git** - это система (программа) контроля версий. Предположим, мы разрабатываем веб-сайт (или любой другой проект). И там приходится постоянно что-то переделывать, изменять, добавлять или удалять. Но однажды, при очередном изменении мы заметили, что проект у нас выдает много ошибок, и всё работает неправильно. Нам необходимо вернуться в то состояние, когда у нас всё работало исправно, потому что иногда лучше начать сначала, чем искать ошибки в многочисленных файлах. В этом нам и помогает программа **Git**. Она, как бы делает "снимок" всех файлов в любой момент времени, в который мы пожелаем. Таким образом, мы можем от каждой точки сохранения повести свою ветку разработки и в любой момент перескакивать на любую из них.

Также отмечу, что данной программой пользуются серьёзные фирмы по программированию. И если вы когда-нибудь туда попадёте, будет большим плюсом знать эту программу. **Git** имеет большое количество команд, но в сегодняшнем уроке мы разберём самые **основы**. И будем разбирать это в ОС WINDOWS.

Для начала необходимо скачать и установить программу <https://git-scm.com/download/win>.

После установки, необходимо указать ваше имя и email. Это необходимо потому, что в каждое ваше сохранение будет записываться автор и email автора. Открываем консоль **Git Bash**. И прописываем:

|  |  |
| --- | --- |
| 1  2 | $ git config --global user.name "Kaluzhskiy Alexsander"  $ git config --global user.email Kaluzhskiy-Alexsander@yandex.ru |

Не забудьте изменить на свои данные!

Проверка настроек:

|  |  |
| --- | --- |
| 1 | $ git config –list |

Выводится:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | core.symlinks=false  core.autocrlf=true  core.fscache=true  color.diff=auto  color.status=auto  color.branch=auto  color.interactive=true  pack.packsizelimit=2g  help.format=html  http.sslcainfo=C:/Program Files/Git/mingw32/ssl/certs/ca-bundle.crt  diff.astextplain.textconv=astextplain  rebase.autosquash=true  credential.helper=manager  user.name=Kaluzhskiy Alexsander  user.email=Kaluzhskiy-Alexsander@yandex.ru |

Отлично! Теперь давайте используем **Git** для сохранения проекта. Необходимо перейти в папку c нашим проектом в **командной строке** Git.

Это можно сделать с помощью досовской команды **cd**(Перейти в папку)

|  |  |
| --- | --- |
| 1 | $ cd «C:/WebServers/home/test2.ru/www/» |

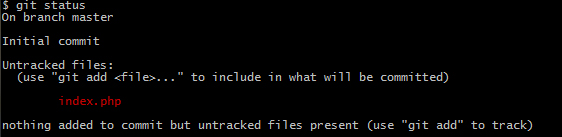
После того как перешли в папку, прописываем команду:

|  |  |
| --- | --- |
| 1 | $ git init |

В нашей папке появится скрытая папка **.git**, где будут находится необходимые программе файлы, чтобы отслеживать наш проект. Эта операция называется **создание нового репозитория**. Значит программа готова работать с нашим проектом.

Следующая команда проверяет статус файлов проекта (которые лежат в нашей директории):

|  |  |
| --- | --- |
| 1 | $ git status |



В начале нам необходимо указать какие именно файлы мы хотим сохранить при нашем сохранении проекта. У меня там лежит один файл **index.php**. Этот файл распечатался красным цветом - значит он пока никак не проиндексирован и не готов к сохранению.

Чтобы это произошло, его необходимо добавить (проиндексировать) с помощью команды **git add**my\_file.

|  |  |
| --- | --- |
| 1 | $ git add index.php |

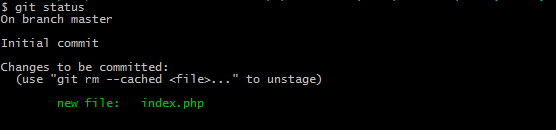
Чтобы добавить несколько файлов, используем команду в таком виде:

|  |  |
| --- | --- |
| 1 | $ git add my\_file1 myfile2 |

Если мы хотим добавить все файлы из директории, используем:

|  |  |
| --- | --- |
| 1 | $ git add . |

Давайте посмотрим команду git status:



Наш файл добавлен и готов к сохранению.

Чтобы удалить файл из отслеживания, используем команду:

|  |  |
| --- | --- |
| 1 | $ git rm --cached [имя\_файла] |

Тогда этот файл не будет попадать в сохранение.

После того, как файлы добавлены с помощью команды add, мы может сделать точку сохранения проекта или по-другому **Коммит**. Вводим команду:

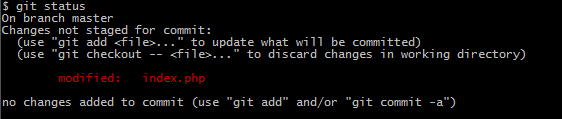
|  |  |
| --- | --- |
| 1 | $ git commit --m "Мой первый коммит" |

Здесь в кавычках пишется пояснение к точке сохранения. Можно писать, что именно было изменено в данном коммите. Если мы сейчас наберём команду git status, то у нас будет:

Команда git status 3

Нечего коммитеть.

Давайте изменим файл **index.php** и посмотрим команду git status



Видим, что **Git** отследил изменения. Чтобы сделать новую точку сохранения, необходимо опять добавить файлы, которые были изменены. Это можно сделать с помощью команды:

|  |  |
| --- | --- |
| 1 | $ git add -u |

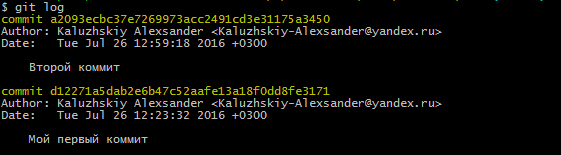
Добавятся изменённые файлы. Можно сделать ещё одну точку сохранения или **коммит**

|  |  |
| --- | --- |
| 1 | $ git commit -m "Второй коммит" |

У нас такая картина:

   
  
Посмотреть историю коммитов:

|  |  |
| --- | --- |
| 1 | $ git log |



Самое интересное! Как нам вернуться на наш первый коммит ? Если мы хотим вернуть состояние файлов к первому коммиту, то нам можно это сделать командой: git checkout хеш-сумма-коммита

|  |  |
| --- | --- |
| 1 | $ git checkout d12271a5dab2e6b47c52aafe13a18f0dd8fe3171 |

Наши файлы вернулись в состояние 1 коммита. Если же мы хотим перейти обратно, сначала посмотрим имя самого последнего коммита с помощью команды **git log --all** (Если мы будет использовать просто git log, то мы не увидим коммиты которые идут после нашего состояния, в которым мы находимся), а потом пропишем команду **git checkout имя\_последнего\_коммита**.

**Выполнение работы**

1. Изучите раздел «Пояснение к работе».
2. Выполните задание.
3. Ответьте на контрольные вопросы.

**Содержание отчета**

Отчет по практической работе должен содержать:

1. Цель работы.
2. Раздел с заданием, в котором расписывается о том, каким образом студент выполнил его, с использованием каких средств, инструментов и технологий. Представленная информация подтверждается скриншотами экрана, файлами, полученными в ходе выполнения практической работы.
3. Ответы на контрольные вопросы.

**Контрольные вопросы**

1. Что такое git brash?
2. Какие команды вы использовали для выполнения задания? Для чего они предназначены?

**Список использованных источников**

**Основные источники:**

1. Федорова Г.Н. Сопровождение информационных систем: учебник для студ. учреждений сред.проф. образования – М.: Издательский центр «Академия», 2018. – 318 с.
2. Федорова Г. Осуществление интеграции программных модулей. Учебник / М.:Академия, 2019 г., -288 с.
3. Боровская Е. В. Основы искусственного интеллекта - М.: Бином. Лаборатория знаний, 2016. – 130 с.

**Дополнительные источники:**

1. Гагарина, Л. Г. Технология разработки программного обеспечения: учеб. пособие / Л. Г. Гагарина, Е. В. Кокорева, Б. Д. Виснадул; Под ред. Л. Г. Гагариной. - М.: ФОРУМ: ИНФРА-М, 2017.-400 с.
2. Чакон Скотт. Pro git / М.: [Springer](https://www.logobook.ru/prod_list.php?ftype=2&par1=10000056&name=Springer&page=1), 2014 г. – 350 с.
3. ГОСТ 24.103-84. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Автоматизированные системы управления. Общие положения
4. ГОСТ 24.104-85 Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Автоматизированные системы управления. Общие требования
5. ГОСТ 24.202-80. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Требования к содержанию документа «Технико-экономическое обоснование»
6. ГОСТ 24.203-80 Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Требования к содержанию общесистемных документов
7. ГОСТ 24.204-80. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Требования к содержанию документа «Описание постановки задачи»
8. ГОСТ 24.205-80 Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Требования к содержанию документов по информационному обеспечению
9. ГОСТ 24.206-80 Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Требования к содержанию документов по техническому обеспечению
10. ГОСТ 24.207-80 Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Требования к содержанию документов по программному обеспечению
11. ГОСТ 24.208-80 Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Требования к содержанию документов стадии «Ввод в эксплуатацию»
12. ГОСТ 24.209-80 Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Требования к содержанию документов по организационному обеспечению
13. ГОСТ 24.210-82 Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Требования к содержанию документов по функциональной части
14. ГОСТ 24.211-82 Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Требования к содержанию документа «Описание алгоритма»
15. ГОСТ 24.301-80 Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Общие требования к выполнению текстовых документов
16. ГОСТ 24.302-80 Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Общие требования к выполнению схем
17. ГОСТ 24.304-82 Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Требования к выполнению чертежей
18. ГОСТ 24.703-85 Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Типовые проектные решения. Основные положения
19. ГОСТ 34.201-89. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Виды, комплектность и обозначение документов при создании автоматизированных систем
20. ГОСТ 34.320- 96 Информационные технологии. Система стандартов по базам данных. Концепции и терминология для концептуальной схемы и информационной базы
21. ГОСТ 34.321- 96 Информационные технологии. Система стандартов по базам данных. Эталонная модель управления данными
22. ГОСТ 34.601 – 90 Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Стадии создания.
23. ГОСТ 34.602-89. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Техническое задание на создание автоматизированной системы
24. ГОСТ 34.603-92. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Виды испытаний автоматизированных систем
25. ГОСТ 6.01.1-87. Единая система классификации и кодирования технико-экономической информации
26. Стандарт ISO/IEC 12207:1995 «InformationTechnology — SoftwareLifeCycleProcesses» (информационные технологии – жизненный цикл программного обеспечения), ГОСТ Р ИСО/МЭК 12207-99.
27. ГОСТ Р ИСО/МЭК 15288-2005. Системная инженерия. Процессы жизненного цикла систем
28. ГОСТ Р ИСО/МЭК ТО 16326-2002. Программная инженерия. Руководство по применению ГОСТ Р ИСО/МЭК 12207 при управлении проектом
29. ISO 10014. Управление качеством — Указания по получению финансовых и экономических выгод.