**МИНИСТЕРСТВО ПРОМЫШЛЕННОСТИ И ТОРГОВЛИ**

**ТВЕРСКОЙ ОБЛАСТИ**

Государственное бюджетное профессиональное образовательное учреждение

«Вышневолоцкий колледж»

**ОТЧЕТ ПО УЧЕБНОЙ ПРАКТИКЕ**

На тему «»

Выполнил студент группы П-46-18

Бибилов Давид Олегович

Руководитель проекта:

Яковлева Жанна Сергеевна

Г. Вышний Волочек

2021 г.

1 Работа в MS Excel, как в реляционные базы данных.

1.1 Разработка простейшей базы данной в программе MS Excel

1.2 Функции ВПР и ГПР, операции с диапазонами

2. Руководство пользователя по работе с GitHub

3 Разработка базы данных

3.1 Моделирование данных. ERD

3.2 Создание базы данных по средствам запросов

3.3 Создание аналитических запросов

4 Разработка приложения для работы в день С-1 (руководство пользователя)

4.1 База данных

4.2 Разработка форм авторизации и поиска

4.3 Разработка форм фильтрации и сортировки

5 Разработка desktop приложений

5.2 Разработка ERD диаграмм

5.2.1 Работа в MS Visio

5.2.2 Подготовка данных для импорта

5.2.3 Разработка базы данных «Мебельная фабрика»

**5.3 Создание приложения**

**5.4 Разработка формы авторизации**

**1 Работа в MS Excel, как в реляционные базы данных.**

**1.1 Разработка простейшей базы данной в программе MS Excel**

**1.2 Функции ВПР и ГПР, операции с диапазонами**

**2. Руководство пользователя по работе с GitHub**

**Введение**

Git — это набор консольных утилит, которые отслеживают и фиксируют изменения в файлах (чаще всего речь идет об исходном коде программ, но вы можете использовать его для любых файлов на ваш вкус). С его помощью вы можете откатиться на более старую версию вашего проекта, сравнивать, анализировать, сливать изменения и многое другое. Этот процесс называется контролем версий. Существуют различные системы для контроля версий. Вы, возможно, о них слышали: SVN, Mercurial, Perforce, CVS, Bitkeeper и другие.

Git является распределенным, то есть не зависит от одного центрального сервера, на котором хранятся файлы. Вместо этого он работает полностью локально, сохраняя данные в папках на жестком диске, которые называются репозиторием. Тем не менее, вы можете хранить копию репозитория онлайн, это сильно облегчает работу над одним проектом для нескольких людей. Для этого используются сайты вроде github и bitbucket.

Git – это распределенная система контроля версий и управления исходным кодом с упором на скорость. Первоначально Git был разработан и разработан Линусом Торвальдсом для разработки ядра Linux. Git – это бесплатное программное обеспечение, распространяемое в соответствии с условиями GNU General Public License версии 2.

**Установка Git**

1. Запустите скачанный на предыдущем шаге exe-файл. Дождитесь появления экрана установки.

2. На первом экране вам предложат согласиться с условиями лицензии GNU GPL. Внимательно их прочитайте, после чего нажмите кнопку Next (как показано на скриншоте):

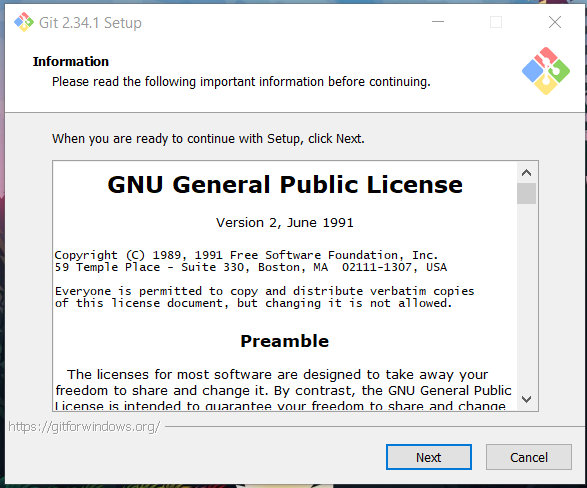
****

Рисунок 1. Условия лицензии

3. Выберите путь для установки Git (лучше его оставить по умолчанию) и нажмите кнопку Next (как показано на скриншоте):

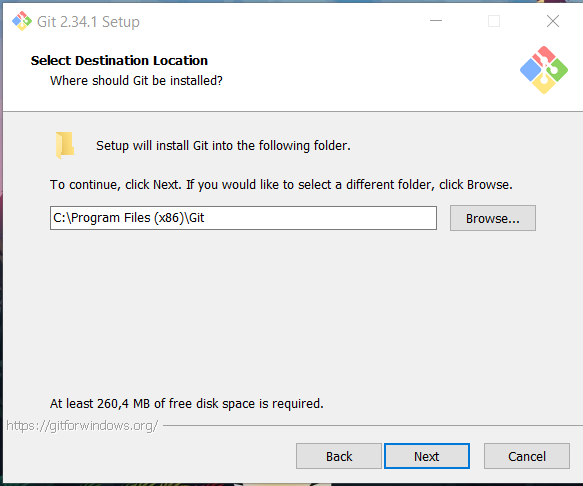
****

Рисунок 2. Путь установки

4. Удостоверьтесь, что выбранные опции (флажки) соответствуют приведённым на скриншоте (они выбраны по умолчанию) и нажмите кнопку Next (как показано на скриншоте):

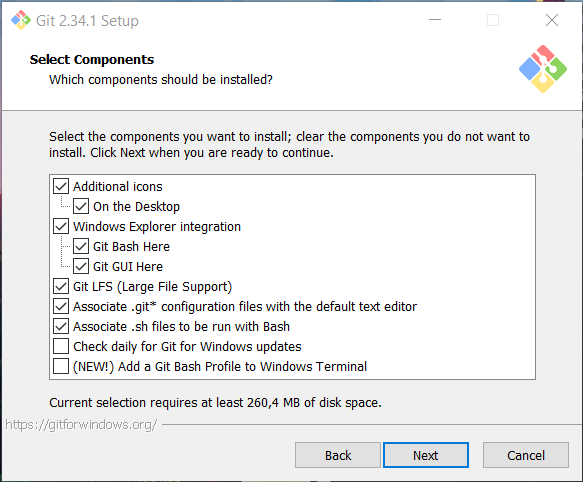
****

Рисунок 3. Компоненты для установки

5. Оставьте значение по умолчанию Git и нажмите кнопку Next (как показано на скриншоте):

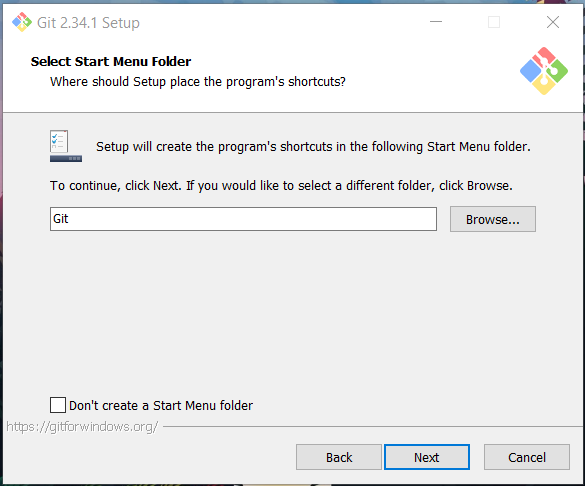
****

Рисунок 4. Имя пункта меню в Пуск

6. Выбранный по умолчанию редактор (Vim) достаточно тяжёл для новичков, поэтому выберите из выпадающего списка опцию Nano Editor и нажмите кнопку Next (как показано на скриншоте):

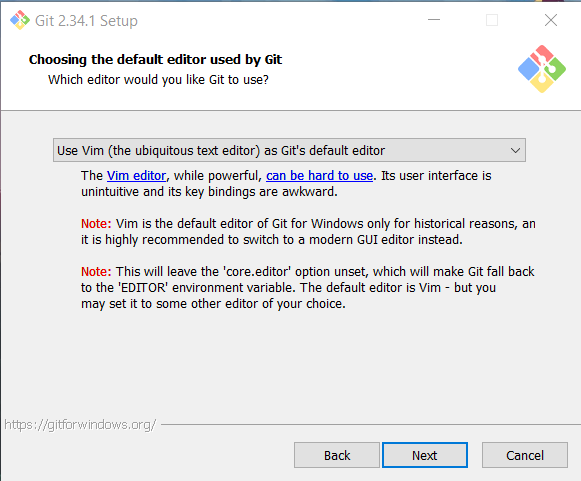
****

Рисунок 5. Редактор по умолчанию

7. На данном этапе необходимо выбрать, добавлять ли Git в переменную окружения PATH. Это набор путей файловой системы, в которой ищутся запускаемые файлы. Если для вас это звучит не понятно - не расстраивайтесь, эта информация нам не особо нужна. Выберите опцию Windows Promt и нажмите кнопку Next (как показано на скриншоте):

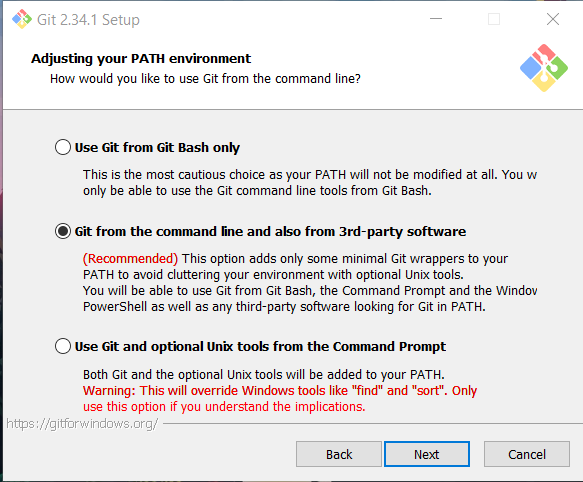
****

Рисунок 6. Переменная окружения PATH

8. Необходимо выбрать библиотеку, которая будет использована для HTTPS-соединений. Оставьте выбранной опцию OpenSSL и нажмите кнопку Next (как показано на скриншоте):

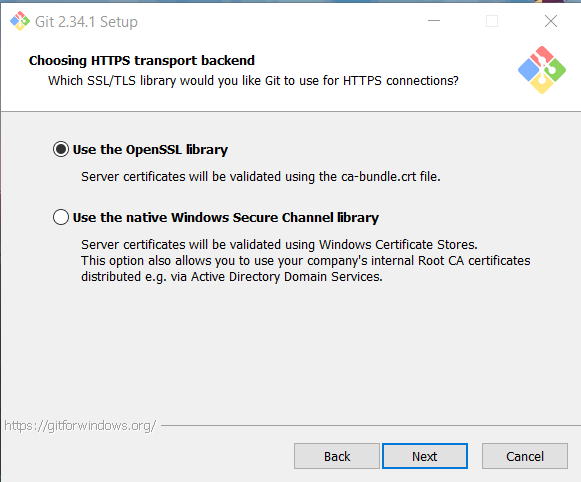
****

Рисунок 7. HTTPS

9. Символы, обозначающие окончание строки различаются в Windows и Unix-подобных ОС (Mac OS, Linux, FreeBSD), поэтому выберите опцию Checkout Windows-style, commit Unix-style line-endings и нажмите кнопку Next (как показано на скриншоте):

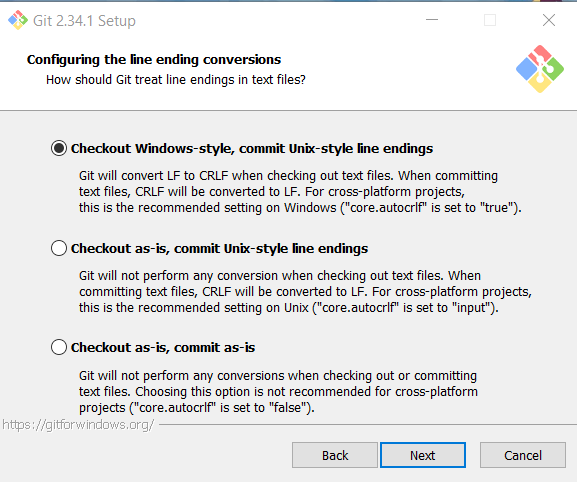
****

Рисунок 8. Символы окончания строки

10. На данном экране вам предлагают выбрать какой терминал (командную строку) вы будете использовать с Git. Оставьте выбранной по умолчанию опцию MinTTY и нажмите кнопку Next (как показано на скриншоте):

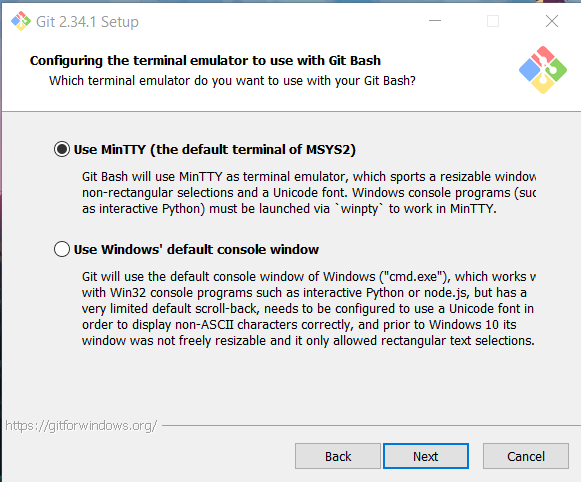
****

Рисунок 9. Терминал

11. Поведение по умолчанию для git pull. Оставьте выбранной опцию Default (fast-forward or merge) и нажмите кнопку Next (как показано на скриншоте):

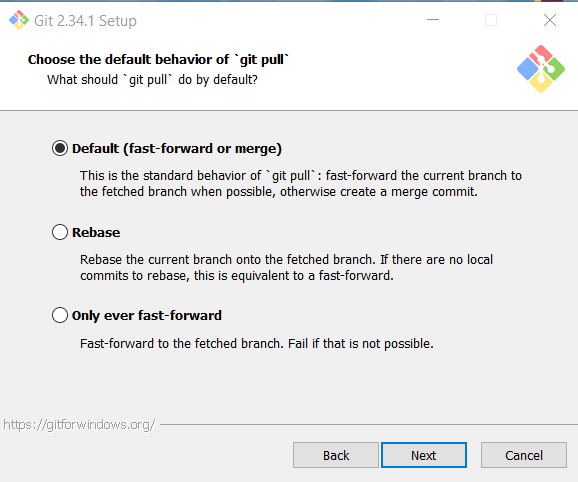
****

Рисунок 10. git pull

12. Выберите значение None и нажмите кнопку Next (как показано на скриншоте):

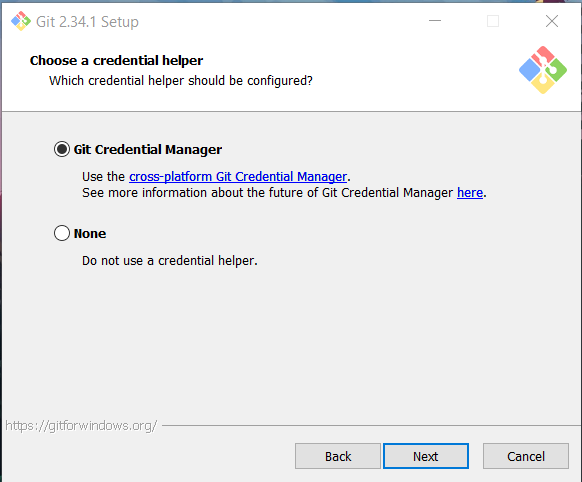
****

Рисунок 11. Credential Manager

13. Убедитесь, что установлен флажок только на Enable file system caching и нажмите кнопку Next (как показано на скриншоте):

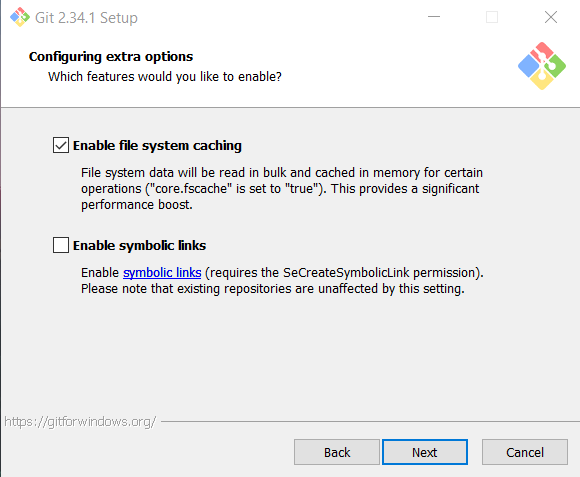
****

Рисунок 12. Extra-опции

14. Убедитесь, что все экспериментальные опции отключены и нажмите кнопку Install (как показано на скриншоте):

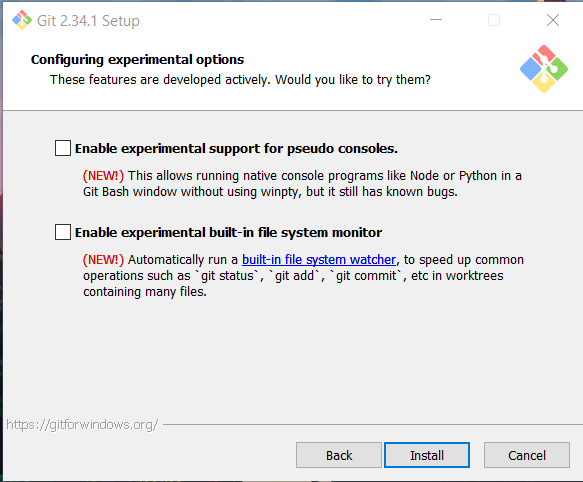
****

Рисунок 13. Экспериментальные опции

15. Процесс установки git:

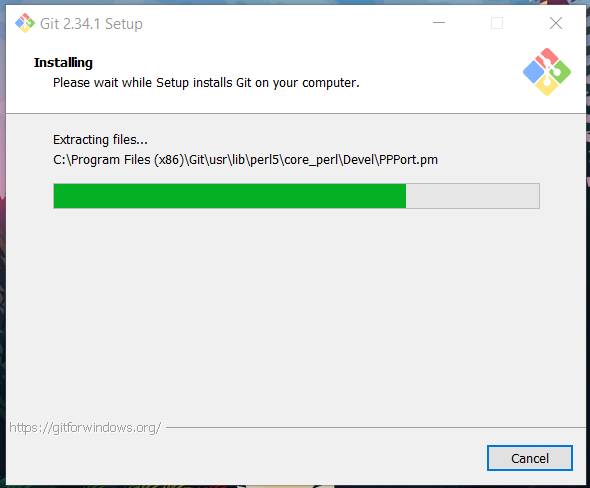
****

Рисунок 14. Процесс установки git:

16. Дождитесь завершения установки и нажмите кнопку Next (как показано на скриншоте):

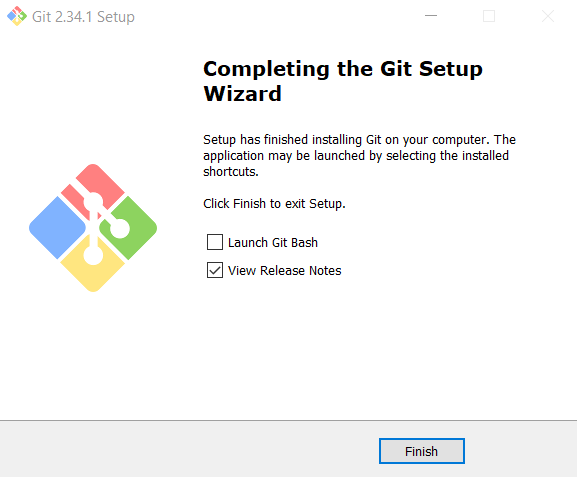
****

Рисунок 15. Завершение установки

**Выгрузка в Git**

1. Проверить наличие программы на компьютере



Рисунок 16. Git

Регистрируемся в Gogs. После регистрации, открывается Панель управления в которой нужно добавить репозиторий

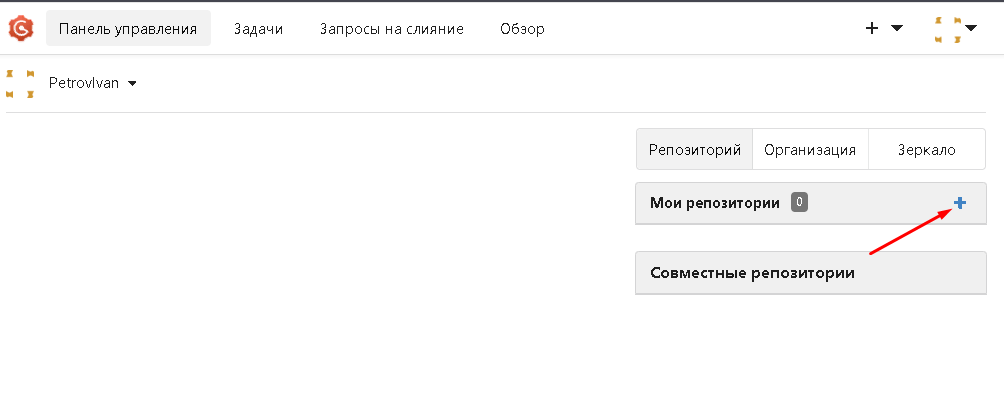


Рисунок 17. Панель управления

1. Имя репозитория First. Видимость ставим галочку Личный репозиторий. Описание обязательно прописываем, например: «Первая программа».

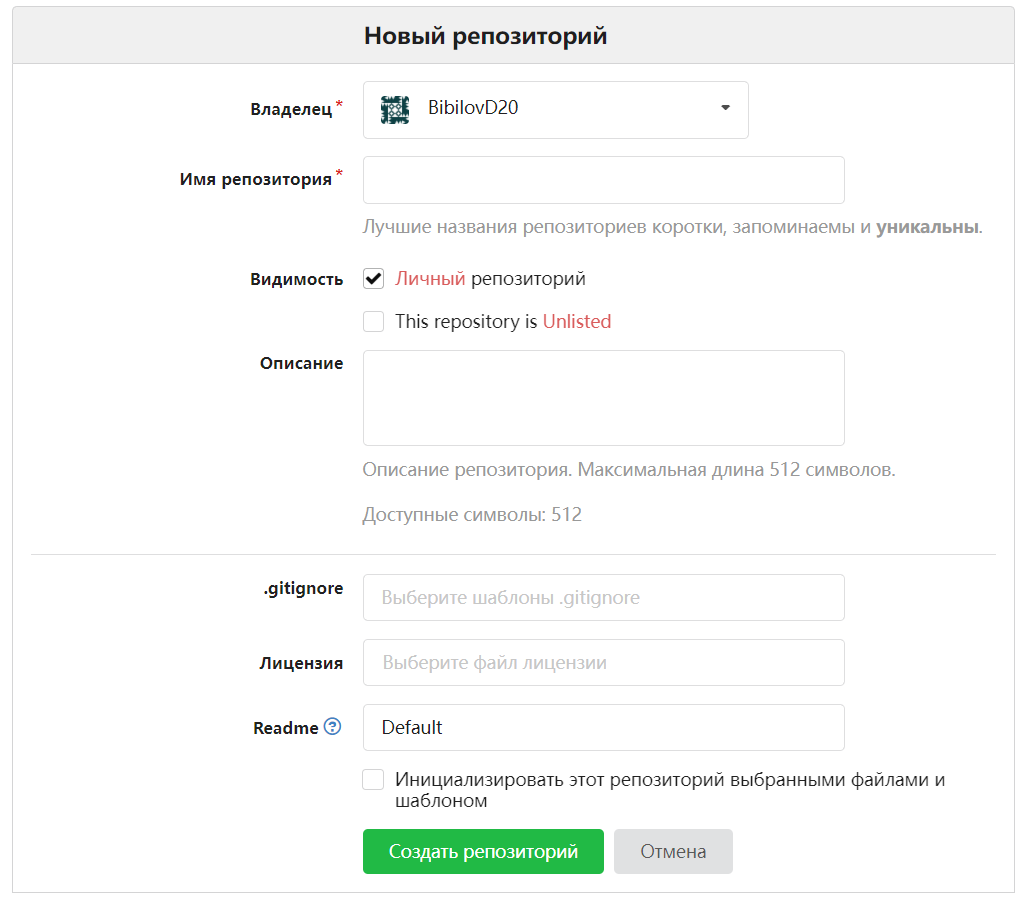


Рисунок 18. Новый репозиторий

1. Появляется краткое руководство, которое позже нам понадобится.
2. Открываем Git Bash Here



Рисунок 19. Git Bash Here

1. Открывается командная строка

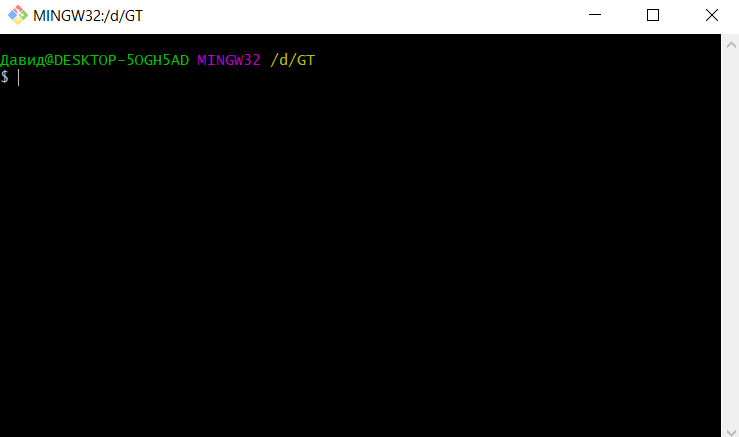


Рисунок 20. Командная строка

1. Используем Краткое руководство из Gogs перестраивая под себя

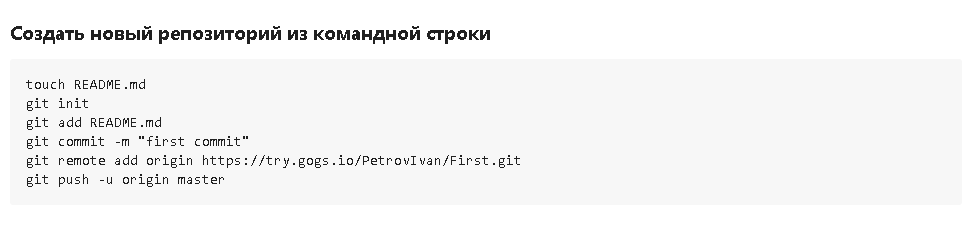


Рисунок 21. Новый репозиторий из командной строки

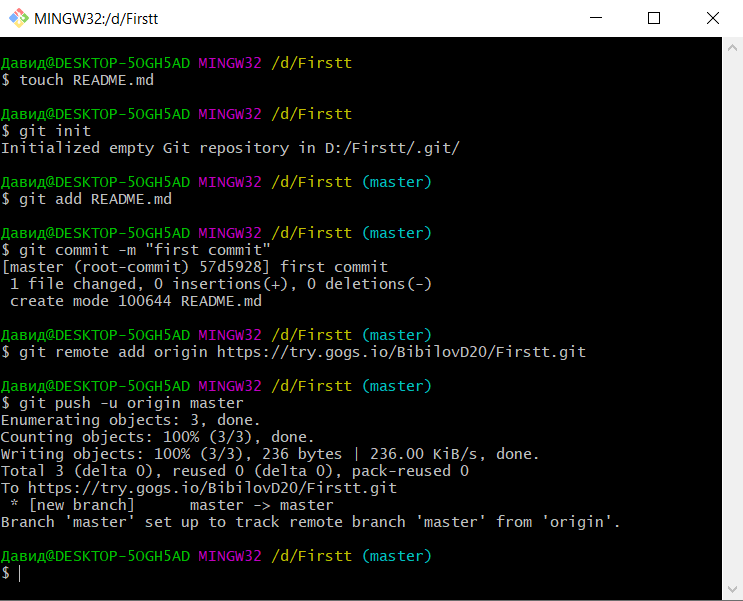


Рисунок 22. Создание репозитория

1. Git config –global user.email “электронная почта на которую зарегистрирован Gogs”

Git config –global user.Name «свое имя»

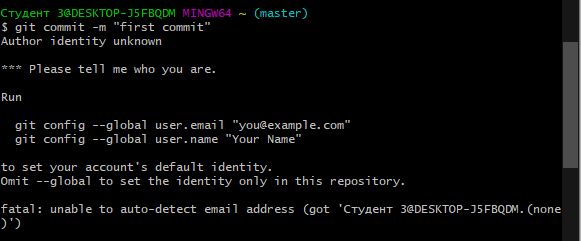


Рисунок 23. Авторизация

1. Должно получиться следующее

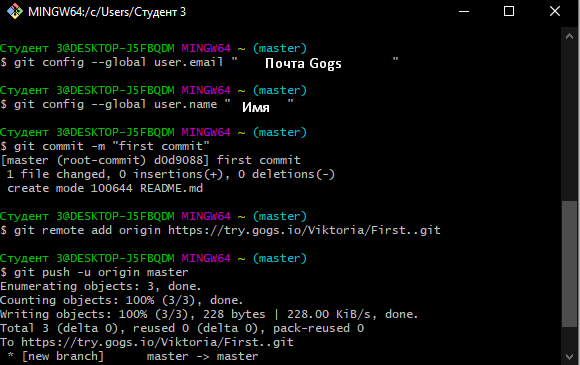


Рисунок 24. Выгрузка в ветвь master

1. Появился Первый репозиторий

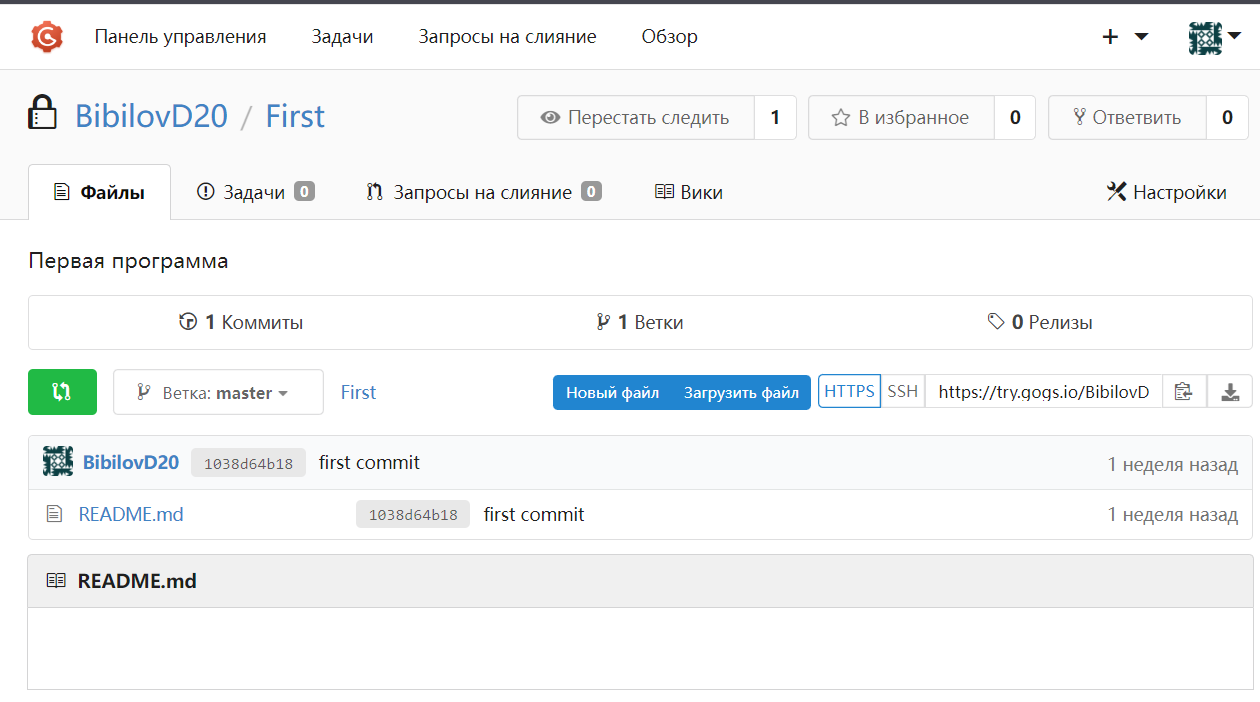


Рисунок 25. Первый репозиторий

1. Создадим отдельную ветку

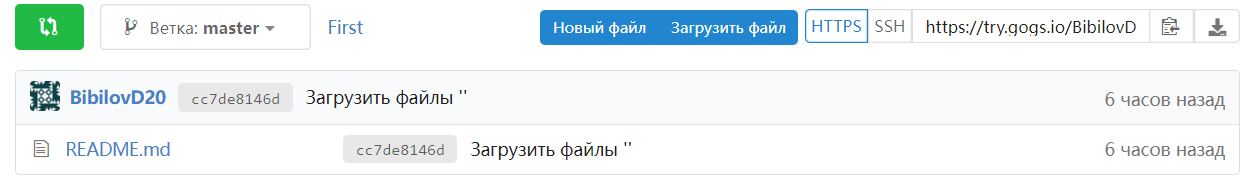


Рисунок 26. Создание ветки

1. Добавляем файл (Например: README.txt), выбираем пункт Создать новую ветвь и называем. Например: too

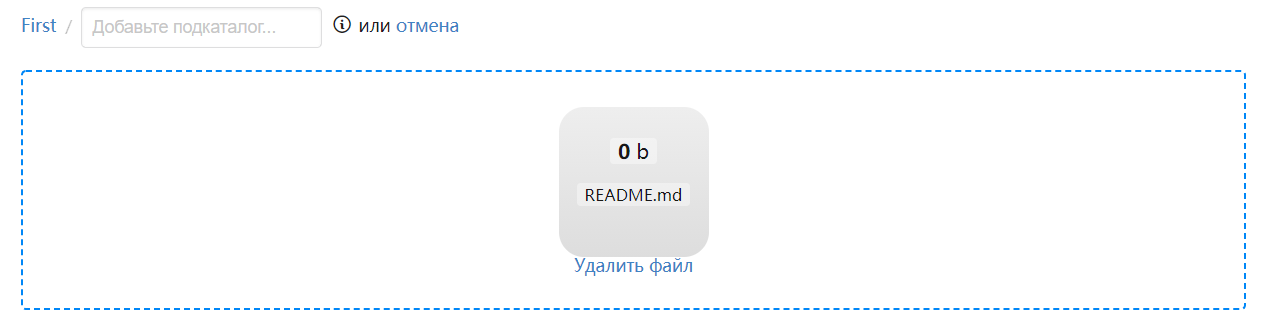


Рисунок 27. Добавление файла «README»

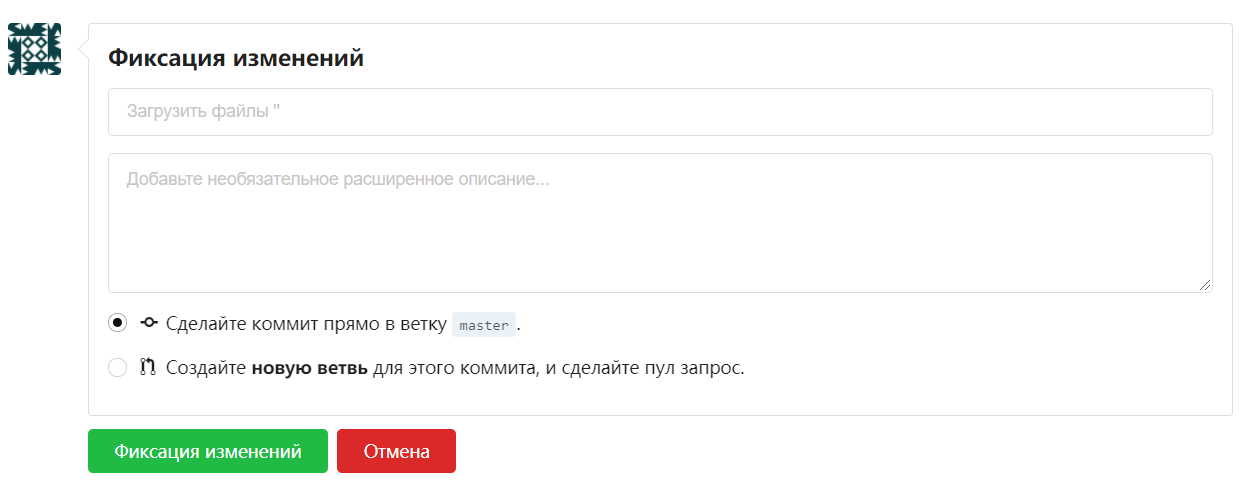


Рисунок 28. Фиксация изменений

1. Добавляем описание и нажимаем Фиксация изменений
2. Возвращаемся в репозиторий и видим, что новая ветка добавлена

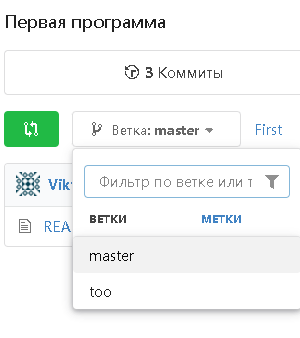


Рисунок 29. Ветки

**4. Команды Git**

* отслеживаемые файлы - когда файл добавляется в отслеживание Git'а создается специальный кэш, благодаря которому можно будет откатываться к предыдущим версиям файла
* не отслеживаемые файлы - когда файл не добавляется в отслеживание Git'а его кэш не хранится в Git'е, а значит откат на предыдущие версии не работает для таких файлов
* коммит - специальное название сохранения версии. Таким образом мы называем все изменения в файлах одним именем и сможем перемещаться между коммитами
* откат - переход на предыдущий коммит и отмена изменений с текущего коммита до того, на который мы хотим вернуться
* пуш - отправка изменений на удаленный репозиторий (в Github, Gitlab, BitBucket)
* фетч - скачивание изменений из репозитория
* ветка - ветки создают, когда нужно несколько активных вариантов разработки. В основном их используют для того, чтобы быстро разработать некоторый функционал, а потом слить ветки в одну
* мердж - слитие веток в одну

Прежде всего нам понадобятся команды для работы с файлами. Знать add конечно хорошо, но нужно иногда и удалять что-то из кэшируемых файлов или изменять поведение Git для определенных файлов. Сегодня я разберу некоторые команды, которые позволят вам манипулировать с файлами, ветками, коммитами в Git.

Init - Данная команда инициализирует систему контроля версий. Для того чтобы создать новый репозиторий достаточно просто ввести:

git init

clone - Данная команда нужна для того, чтобы клонировать репозиторий из облака. Можно клонировать репозитории из Github, Gitlab, BitBucket и других сервисов. Для того чтобы склонировать репозиторий нужно ввести:

git clone ссылка-на-репозиторий

add - Данная команда понадобится вам, когда вам нужно добавить файл для кэширования. Давайте разберёмся как это работает:

Вы изменяете файлы, можете изменять достаточно много файлов, задачи изменения которых вообще никак не связаны

Вы решаете "закоммитить" ваши файлы (сделать сохранение версии, для того чтобы Git запомнил все ваши изменения в файлах и как-то назвал их, для этого есть отдельная команда git commit)

Вы также можете добавить файлы, которые ранее не отслеживались, для того чтобы Git занёс их в свою систему хранения версий и вы могли откатываться на какую-то из версий файла

Обычно с самого начала все разработчики делают коммит (сохранение версии) с названием init, занося все файлы в систему отслеживания. Делается это с помощью простой последовательности команд:

git add . # Занесение всех файлов для отслеживания

git commit -m "init" # Создание версии

Проделав данный алгоритм, состоящий из двух команд вы занесёте все файлы из вашего проекта в систему контроля версий Git.

Но, что если вам не хочется вносить все файлы? Тогда вы может использовать следующий синтаксис:

git add ./file # Добавление файла для отслеживания

git add ./dir # Добавление всех файлов из директории и дочерних директорий для отслеживания

Так, хорошо, но что если нам нужно занести весь проект в систему отслеживания, однако есть пару директорий, которые мы не хотим заносить (например, node\_modules или out)? Для этого мы можем создать файл .gitignore, в котором будут прописаны пути до файлов и директорий, которые Git должен игнорировать:

node\_modules, out, one\_file.tsx - В данном случае Git будет игнорировать директорию node\_modules, out и файл one\_file.tsx

Также мы можем сделать так, чтобы Git искал некоторые названия в дочерних директориях и игнорировал их:

./\*/\*\*/node\_modules - Данный пример говорит о том, что Git будет игнорировать директорию node\_modules, которая будет находится в любой дочерней директории (включая корневую).

А что если я хочу обновить уже закэшированные файлы, мне что теперь их писать по одному? Конечно же нет. Для этого существует специальный флаг -u (сокращение от --update), который позволяет закэшировать только те файлы, которые уже отслеживаются:

git add -u, rm - Данная команда поможет, когда нам нужно избавиться от файла, она подобно команде rm удаляет файл из файловой системы и кэша Git, позволяя нам быстро избавиться от файла.

git rm file.txt - Данный пример удалит файл file.txt из кэша и файловой системы.

git rm --cached file.txt - Данная команда удалит файл из "кэша", но что это значит? Допустим, что мы "закоммитили" (сохранили версию, об этом поговорим вот уже совсем скоро) наш файл, а теперь хотим, чтобы Git считал что мы его удалили, а сами просто оставили его на диске. Для этого и нужна команда выше. Она просто говорит Git: "Слушай, а давай ты просто удалишь этот файл из кэша и не будешь его показывать в репозитории, а я оставлю копию у себя на ПК, ок?"

Таким образом мы можем работать с данным файлом и Git не будет знать что именно в нём мы изменяем, а затем просто можем опять добавить его. Файл будет висеть в состоянии "untracked" (не отслеживается) до тех пор, покуда мы его опять не добавим.

Commit - Думаю, что стоит поговорить об этом уже сейчас, ибо позже без данной команды уже никуда. Данная команда буквально сохраняет версию, в которой указывается что именно в файлах мы изменили.

Обычно разработчики сохраняют версию программы с помощью данной команды:

git commit -am "Что было сделано" - В данном случае мы просто скажем, чтобы Git сохранил все изменения, которые мы сделали в файлах, которые отслеживаются. Файлы, которые не отслеживаются просто не попадают в версию, а также их изменения никак нельзя отследить.

Для того чтобы отменить последний коммит (отменить не изменения, а именно просто разкоммитить изменения) и совместить его с текущими изменениями используйте команду:

git commit --amend -am "Название коммита"

show - Данная команда нужна для того, чтобы быстро показать что было сделано за предыдущие коммит:

git show

Если до этого момента вы не знали как проверить какой файл кэшируется, какой не отслеживается, а какие файлы и вовсе только что были удалены из кэша, то спешу вас обрадовать, у вас теперь есть команда git status, которая даёт возможность посмотреть что именно происходит с вашими файлами:

git status

git status -v

Если же вам наоборот хочется увидеть бриф по информации, то добавьте флаг -s (сокращение от --short):

git status -s

log

До этого момента мы могли только посмотреть что творилось в предыдущем коммите, однако с помощью git log мы можем посмотреть историю наших коммитов:

git log

Если же вы хотите красиво вывести все ваши коммиты в столбик, то используйте данную команду:

git log --graph --pretty=oneline --abbrev-commit

В данной команде мы описываем что хотим просматривать коммиты и их названия в одну строку, хотим чтобы коммиты сокращали свой ID, а также чтобы всё это красиво было разбито по списку

Diff - С помощью данной команды мы можем посмотреть на изменения между коммитами. Эта команда является одной из самых мощных в Git, вам стоит обратить на неё внимание:

git diff ID-коммита-1 ID-коммита-2 - Так называемые "ID-коммита" можно взять и вышеприведенной git log

Также, можно посмотреть что было изменено с момента последнего коммита (что изменено и попало в кэш) с помощью флага --cached:

git diff –cached - Если вы хотите посмотреть историю изменений в файлах в определенном коммите, то используйте следующую команду:

git diff ID-коммита

Если вы хотите посмотреть на статистику, то вы всегда можете использовать флаг --stat:

git diff --stat ID-коммита

Если вы хотите посмотреть на изменения только тех файлов, которые добавлены для отслеживания, то нужно ввести следующую команду:

git diff --staged

branch

В Git есть ветки для разделения версий. Если коммит нужен для того, чтобы сделать snapshot (слепок изменений) файлов, то ветки нужны для того, чтобы эти snapshot'ы разделять. У вас может быть ветка

master

build

beta

alpha

любое другое название

Для того чтобы перечислить все ветки с помощью Git нужно ввести следующую команду:

git branch

Для того чтобы создать новую ветку нужно ввести:

git branch имя-ветки

checkout

Checkout используют для того, чтобы переходить между ветками. Для того чтобы перейти на другую ветку достаточно просто написать:

git checkout имя-ветки

Для того чтобы создать ветку и сразу же перейти на неё достаточно ввести:

git checkout -b имя-ветки

merge

Соединение веток не являются сложной темой. Для того чтобы слить текущую ветку с другой нужно ввести:

git merge название-ветки-с-которой-будем-мерджить

push - Для того чтобы закинуть изменения на сервер нужна команда push, которая позволит нам закинуть текущую ветку на удаленный репозиторий.

**5. Visual Studio Code**

Данное ПО является хорошим выбором для комфортной работы с GIT и GITHUB. В VS Code есть встроенный терминал, который можно вызвать с помощью комбинации клавиш:

ctrl + `

Внутри данного терминала вы можете использовать те же команды для работы с GIT которые были описаны выше.

Настройка терминала VS Code в Windows

По умолчанию консоль VS Code в Windows запускает стандартную командную строку Windows, которая не может работать с GIT, чтобы все работало нужно запустить терминал bash, для этого выполним следующие действия:

Убедимся, что ранее нами был скачан и установлен git, подробнее данный процесс был описан выше.

Запускаем Visual Studio Code.

Зажимаем сочетание клавиш Ctrl + Shift + P вводим open user setting и из выпадающего меню выбираем Open User Settings.

В параметры поиска вводим terminal.integrated.shell.windows и заменяем cmd.exe на C:\\Program Files\\Git\\bin\\bash.exe (если вы установили git в другую директорию то ссылка будет иметь вид относительно установленной вами директории).

Перезагружаем Visual Studio Code

Открываем терминал и проверяем, что запущен bash терминал

**Алиасы**

Алиас это лишь еще одна, созданная вами команда для git'а, которая позволяет запускать любую другую команду git'а или же внешний по отношению к нему скрипт. Добавлять алиасы можно с помощью утилиты config, имеющейся в git'е. Данная утилита различает три уровня действия задаваемых Вами настроек:

1) Для всех пользователей в системе - $ git config --system

2) Для текущего юзера - $ git config --global

3) Для текущего git репозитория - $ git config

Каждый последующий уровень имеет более высокий приоритет и может переопределять настройки более общих уровней. Большинство алиасов я прописываю для уровня текущего пользователя, под которым и работаю.

Добавляем алиасы:

Давайте рассмотрим алиасы на примере тех, которые я постоянно использую в работе.

- git status

Данная команда употребляется довольно часто и иногда банально лень ее писать, даже с дополнением табуляцией, поэтому для нее у меня появился следующий алиас:

$ git st

который можно добавить с помощью

$ git config --global alias.st status

- git commit

Немногим реже приходится что-то комитить, плюс добавлять понятное описание к комиту, т.е. использовать опцию -m, тут и появились два новых алиаса:

$ git cm 'some desc'

$ git cma 'some desc'

Которые мы добавим следующим образом:

$ git config --global alias.cm 'commit -m'

$ git config --global alias.cma 'commit -a -m'

Первый из них позволяет просто ввести сообщение для коммита, и сам добавит в него все файлы из staging области. Второй добавит в коммит все существующие измененные и удаленные файлы, но не добавит untracked файлы.

- git checkout

Все мы время от времени совершаем ошибки и хотим их быстро исправить, тут мне помогает следующий алиас:

$ git config --global alias.co checkout

- git branch

Раньше я не часто использовал ветки в git'е, но сейчас они стали неотъемлимой частью моей работы, и тут появляется очередной удобный алиас:

$ git config --global alias.nb 'checkout -b'

Этот алиас легко запомнить, и он прост в использовании:

$ git nb 4devs-custom-errors

Просмотр имеющихся алиасов

Чтобы посмотреть на список имеющийхся алиасов, можно использовать

$ git config --list

который помимо всего прочего выведет секцию alias, в моем случае она выглядит примерно так:

alias.st=status

alias.cm=commit -m

alias.co=checkout

alias.nb=checkout -b

**Отмена изменений и возврат к другому коммиту**

1. Отменить коммит, но оставить изменения:

Для того чтобы отменить последний коммит git без удаления изменений используется команда reset с параметром --soft. Команде надо передать идентификатор коммита или его позицию относительно HEAD. В терминологии git термин HEAD - это самая последняя версия проекта в текущей ветке. С помощью HEAD можно ссылаться на коммиты в истории. Для этого используется символ ~. Таким образом ссылка на предыдущий коммит будет выглядеть как HEAD~ или HEAD~1, а на коммит перед ним - HEAD~2 и так далее. Для отмены последнего коммита достаточно выполнить команду:

git reset --soft HEAD~

Если посмотреть отличия HEAD и текущего состояния проекта, то будет видно, добавление файла file3:

git diff HEAD

Аналогичного результата можно добиться, передав идентификатор коммита, например, давайте отменим коммит, добавляющий file2. Для этого посмотрите идентификатор коммита перед ним, в данном случае, это "Inital Commit" с помощью следующей команды:

git log

А затем передайте его в команду git reset. Например:

git reset --soft 887080eea5fd8bd3bc2503dcf043ac6f5c19a8e5

И снова все файлы на месте, а в HEAD теперь будет добавлено два файла: file2 и file3:

Таким образом вы можете отменить несколько коммитов за раз, надо только указать идентификатор самого раннего коммита.

Обратите внимание, что файлы, которые ранее были в коммите, сейчас всё ещё добавлены в индекс, поэтому вам не надо вызывать git add, можно сразу создавать новый коммит. Но у команды reset есть ещё одна опция: --mixed. Она используется по умолчанию. При использовании этой опции ваши изменения тоже сохраняются, но перед следующим коммитом их снова надо будет добавить в индекс с помощью git add. При выполнении команды git status эти файлы будут отображаться как не отслеживаемые:

git status

2. Отменить коммит и удалить изменения:

Отмена коммита git с удалением изменений работает аналогично. Только здесь необходимо вместо опции --soft указывать опцию --hard. Например, при той же структуре коммитов, можно удалить последний коммит с добавлением файла file3 вместе с этим файлом:

git reset --hard HEAD~

Теперь файла нет. Аналогично, вы можете указать идентификатор коммита, до которого надо отменить коммиты. Обратите внимание, что указывается не тот коммит, который надо отменить, а коммит перед ним. Ещё важно отметить, что это всё работает пока вы не отправили свои коммиты в удалённый репозиторий. Если коммиты уже отправлены, их идентификаторы сохранены там, а поэтому менять их нельзя, иначе могут возникнуть конфликты слияния, которые будет сложно решить. Теперь вы знаете отменить последний локальный коммит git.

3. Вернуть отмененный коммит:

Если вы всё же удалили что-то нужное с помощью команды reset --hard, и вовремя об этом вспомнили, то можно попытаться вернуть потерянные данные. Первый способ будет работать если вы ещё ничего не комитили после отмены комита. Для того чтобы посмотреть историю добавления/удаления коммитов используйте команду: git reflog

Затем, для того чтобы вернуться к нужному удалённому коммиту надо использовать ту же команду reset --hard со ссылкой на удалённый коммит, полученной из предыдущей команды, в виде HEAD{@номер}. Например, для коммита c файлом file3 это будет выглядеть так:

git reset --hard HEAD@{1}

Аналогично можно использовать адрес: git reset --hard fc1f295

Срок хранения удалённых коммитов ограничен. Время от времени git удаляет мусор, так что если ждать слишком долго, то нужных данных уже может и не быть. Но найти удалённые коммиты, если git их ещё не удалил можно с помощью такой команды: git fsck --lost-found

Затем просто используйте идентификатор коммита для того чтобы посмотреть какие в нём были изменения:

git show 8a996dd76fbacb05a2df91c0f2d19b1a3afd8451

Затем можно переключиться на этот коммит с помощью команды:

git rebase 8a996dd76fbacb05a2df91c0f2d19b1a3afd8451

Это всё тоже безопасно делать только с коммитами, ещё не отправленными в удалённый репозиторий.

4. Отменить изменения, но не менять коммит:

Если вы уже отправили коммит в удалённый репозиторий, удалять его не желательно, потому что если кто-то успеет скачать репозиторий до отмены коммита, то потом у него возникнут проблемы. Однако изменения, сделанные в коммите можно отменить новым коммитом. Для того чтобы не делать это вручную существует специальная команда revert. Её уже нужно передать идентификатор именно того коммита, изменения из которого надо отменить. Для этого сначала найдите хэш коммита:

git log

Затем выполните команду revert, например:

git revert 8a996dd76fbacb05a2df91c0f2d19b1a3afd8451

Команда предложит вам написать сообщение для отменяющего коммита, можно просто закрыть этот файл:

Затем изменения, которые были в коммите исчезнут и уже это можно будет снова пушить в удалённые репозиторий.

**Работа с ветками Git branch**

Управление ветками. Теперь, когда вы уже попробовали создавать, объединять и удалять ветки, пора познакомиться с некоторыми инструментами для управления ветками, которые вам пригодятся, когда вы начнёте использовать ветки постоянно.

Команда git branch делает несколько больше, чем просто создаёт и удаляет ветки. При запуске без параметров, вы получите простой список имеющихся у вас веток:

$ git branch

iss53

\* master

testing

Обратите внимание на символ \*, стоящий перед веткой master: он указывает на ветку, на которой вы находитесь в настоящий момент (т. е. ветку, на которую указывает HEAD). Это означает, что если вы сейчас сделаете коммит, ветка master переместится вперёд в соответствии с вашими последними изменениями. Чтобы посмотреть последний коммит на каждой из веток, выполните команду git branch -v:

$ git branch -v

iss53 93b412c Fix javascript issue

\* master 7a98805 Merge branch 'iss53'

testing 782fd34 Add scott to the author list in the readme

Опции --merged и --no-merged могут отфильтровать этот список для вывода только тех веток, которые слиты или ещё не слиты в текущую ветку. Чтобы посмотреть те ветки, которые вы уже слили с текущей, можете выполнить команду git branch --merged:

$ git branch --merged

iss53

\* master

Ветка iss53 присутствует в этом списке, потому что вы ранее слили её в master. Те ветки из этого списка, перед которыми нет символа \*, можно смело удалять командой git branch -d; наработки из этих веток уже включены в другую ветку, так что ничего не потеряется.

Чтобы увидеть все ветки, содержащие наработки, которые вы пока ещё не слили в текущую ветку, выполните команду git branch --no-merged:

$ git branch --no-merged

Testing

Вы увидите оставшуюся ветку, так как она содержит ещё не слитые наработки, попытка удалить её командой git branch -d приведёт к ошибке:

$ git branch -d testing

error: The branch 'testing' is not fully merged.

If you are sure you want to delete it, run 'git branch -D testing'.

Если вы действительно хотите удалить ветку вместе со всеми наработками, используйте опцию -D, как указано в подсказке.

Подсказка

Если в качестве аргумента не указан коммит или ветка, то опции --merged и --no-merged покажут что уже слито или не слито с вашей текущей веткой соответственно.

Вы всегда можете указать дополнительный аргумент для вывода той же информации, но относительно указанной ветки предварительно не извлекая и не переходя на неё.

$ git checkout testing

$ git branch --no-merged master

topicA

featureB

Переименование ветки

Внимание

Не переименовывайте ветки, которые всё ещё используются другими участниками. Не переименовывайте ветку в master/main/mainline, не прочитав раздел «Изменение имени главной ветки».

Предположим, у вас есть ветка с именем bad-branch-name, и вы хотите изменить её на corrected-branch-name, сохранив при этом всю историю. Вместе с этим, вы также хотите изменить имя ветки на удалённом сервере (GitHub, GitLab или другой сервер). Как это сделать?

Переименуйте ветку локально с помощью команды git branch --move:

$ git branch --move bad-branch-name corrected-branch-name

Ветка bad-branch-name будет переименована в corrected-branch-name, но это изменение пока только локальное. Чтобы все остальные увидели исправленную ветку в удалённом репозитории, отправьте её туда:

$ git push --set-upstream origin corrected-branch-name

Теперь проверим, где мы сейчас находимся:

$ git branch --all

\* corrected-branch-name

main

remotes/origin/bad-branch-name

remotes/origin/corrected-branch-name

remotes/origin/main

Обратите внимание, что текущая ветка corrected-branch-name, которая также присутствует и на удалённом сервере. Однако, старая ветка всё ещё по-прежнему там, но её можно удалить с помощью команды:

$ git push origin --delete bad-branch-name

Теперь старое имя ветки полностью заменено исправленным.

Изменение имени главной ветки

Предупреждение

Изменение имени ветки, например master/main/mainline/default, сломает интеграции, службы, вспомогательные утилиты и скрипты сборки, которые использует ваш репозиторий. Прежде чем сделать это, обязательно проконсультируйтесь с коллегами. Также убедитесь, что вы выполнили тщательный поиск в своём репозитории и обновили все ссылки на старое имя ветки в вашем коде или скриптах.

Переименуйте локальную ветку master в main с помощью следующей команды:

$ git branch --move master main

После этого, локальной ветки master больше не существует, потому что она была переименована в ветку main.

Чтобы все остальные могли видеть новую ветку main, вам нужно отправить её в общий репозиторий. Это делает переименованную ветку доступной в удалённом репозитории.

$ git push --set-upstream origin main

В итоге, состояние репозитория становится следующим:

git branch --all

\* main

remotes/origin/HEAD -> origin/master

remotes/origin/main

remotes/origin/master

Ваша локальная ветка master исчезла, так как она заменена веткой main. Ветка main доступна в удалённом репозитории. Старая ветка master всё ещё присутствует в удалённом репозитории. Остальные участники будут продолжать использовать ветку master в качестве основы для своей работы, пока вы не совершите ряд дополнительных действий.

Теперь, для завершения перехода на новую ветку перед вами стоят следующие задачи:

Все проекты, которые зависят от текущего, должны будут обновить свой код и/или конфигурацию.

Обновите конфигурацию всех запускаемых тестов.

Исправьте скрипты сборки и публикации артефактов.

Поправьте настройки репозитория на сервере: задайте новую ветку по умолчанию, обновите правила слияния, а также прочие настройки, которые зависят от имени веток.

Обновите документацию, исправив ссылки, указывающие на старую ветку.

Слейте или отмените запросы на слияние изменений, нацеленные на старую ветку.

После того, как вы выполнили все эти задачи и уверены, что ветка main работает так же, как ветка master, вы можете удалить ветку master:

$ git push origin --delete master

**Сохранение фрагментов кода, создание Gist**

Для тех, кто не знаком с этим сервисом — Gist предназначен для обмена кусочками кода отдельных программ между людьми. В основе Gist лежит система контроля версий git, что делает возможным коллективную работу над материалом. Вы можете оставлять для себя заметки, делиться отдельными файлами, фрагментами кода или полными приложениями.

Размещенный в Gist код можно:

* скачать в виде отдельного файла
* комментировать
* клонировать в свой репозиторий.

Создадим новый фрагмент кода - жмем на кнопку New gist.

На следующей странице откроется текстовый редактор для написания кода. Жмем на кнопку Create Public Gist и сохраняем наш фрагмент кода.

После сохранения-публикации мы можем:

* Edit - отредактировать код
* Download - скачать фрагмент в виде файла с правильным расширением
* Star - отметить фрагмент как понравившийся
* Raw - просмотреть код в виде простого текста
* Delete - удалить фрагмент кода

Доступ к сервису Gist.Github имеют все зарегистрировавшиеся пользователи и, естественно, этот сервис для публичного использования предоставляется бесплатно.

С помощью этого сервиса можете поделиться каким-либо фрагментом кода с друзьями, например в Твиттере, хранить и изменять какие-либо фрагменты кода (при этом каждому изменению будет присваиваться своя версия)

**Работа с панелью Система управления версиями в редакторе кода VS Code**

1. Вкладка Source Control

Прежде всего, чтобы воспользоваться преимуществами интеграции контроля исходного кода, следует инициализировать проект как репозиторий Git.

Откройте Visual Studio Code и запустите встроенный терминал. Вы можете открыть его, используя сочетание клавиш CTRL + ` в Linux, macOS или Windows.

Используя терминал, создайте каталог для нового проекта и перейдите в этот каталог:

mkdir git\_test

cd git\_test

Затем создайте репозиторий Git:

git init

Также вы можете сделать это в Visual Studio Code, открыв вкладку Source Control (иконка выглядит как развилка дороги) в левой панели:

Иконка Source Control

Затем нажмите кнопку Open Folder:

Снимок экрана с кнопкой Open Folder

При нажатии кнопки откроется проводник файлов, где будет открыт текущий каталог. Выберите предпочитаемый каталог проекта и нажмите Open.

Затем нажмите Initialize Repository:

Снимок экрана с кнопкой инициализации репозитория Initialize Repository

Если теперь вы посмотрите на свою файловую систему, вы увидите, что она содержит каталог .git. Чтобы сделать это, используйте терминал для перехода в каталог проекта и вывода его содержимого:

ls -la

Вы увидите созданный каталог .git:

Output

.git

Это означает, что репозиторий инициализирован, и теперь вам следует добавить в него файл index.html.

После этого на панели Source Control вы увидите, что рядом с именем вашего нового файла отображается буква U. Обозначение U означает, что файл не отслеживается, то есть, что это новый или измененный файл, который еще не был добавлен в репозиторий:

Снимок экрана с изображением неотслеживаемого файла, обозначенного буквой U

Вы можете нажать значок плюс (+) рядом с файлом index.html, чтобы включить отслеживание файла в репозитории.

После этого рядом с файлом появится буква A. A обозначает новый файл, который был добавлен в репозиторий.

Чтобы записать изменения, введите команду отправки в поле ввода в верхней части панели Source Control. Затем нажмите иконку отметки check для отправки файла в репозиторий.

Снимок экрана с изображением добавленного файла, помеченного буквой A, и команды отправки

После этого вы увидите, что несохраненных изменений нет.

Теперь добавьте немного содержания в файл index.html.

Вы можете использовать ярлык Emmet для генерирования базовой структуры кода HTML5 в VS Code, нажав !, а затем клавишу Tab. Теперь добавьте что-нибудь в раздел <body>, например, заголовок <h1>, и сохраните файл.

На панели исходного кода вы увидите, что ваш файл изменился. Рядом с именем файла появится буква M, означающая, что файл изменен:

Снимок измененного файла, обозначенного буквой M

Для практики давайте запишем это изменение в репозиторий.

Теперь вы познакомились с работой через панель контроля исходного кода, и мы переходим к интерпретации показателей gutter.

2. Интерпретация показателей Gutter

На этом шаге мы рассмотрим концепцию Gutter («Желоб») в VS Code. Gutter — это небольшая область справа от номера строки.

Если ранее вы использовали сворачивание кода, то в области Gutter находятся иконки «Свернуть» и «Развернуть».

Для начала внесем небольшое изменение в файл index.html, например, изменим содержание внутри тега <h1>. После этого вы увидите, что измененная строка помечена в области Gutter синей вертикальной чертой. Синяя вертикальная черта означает, что соответствующая строка кода была изменена.

Теперь попробуйте удалить строку кода. Вы можете удалить одну из строк в разделе <body> вашего файла index.html. Обратите внимание, что в области Gutter появился красный треугольник. Красный треугольник означает строку или группу строк, которые были удалены.

Теперь добавьте новую строку в конец раздела <body> и обратите внимание на зеленую полосу. Вертикальная зеленая полоса обозначает добавленную строку кода.

В этом примере описаны индикаторы области Gutter для случаев изменения, удаления и добавления строки:

Снимок экрана с примерами трех индикаторов области Gutter

3. Просмотр отличий файлов

VS Code также позволяет посмотреть отличия между разными версиями файла. Обычно для этого нужно загружать отдельный инструмент diff, так что встроенная функция повысит эффективность работы.

Чтобы посмотреть отличия, откройте панель контроля исходного кода и дважды нажмите на измененный файл. В этом случае следует дважды нажать на файл index.html. Откроется типовое окно сравнения, где текущая версия файла отображается слева, а ранее сохраненная в репозитории версия — справа.

В этом примере мы видим, что в текущей версии добавлена строка:

Снимок разделенного экрана для сравнения версий

4. Работа с ветвлением

Вы можете использовать нижнюю панель для создания и переключения ветвей кода. В нижней левой части редактора отображается иконка контроля исходного кода (которая выглядит как дорожная развилка), после которой обычно идет имя главной ветви или ветви, над которой вы сейчас работаете.

Индикатор ветви в нижней панели VS Code с именем: master

Чтобы создать ветвление, нажмите на имя ветви. Откроется меню, где вы сможете создать новую ветвь:

Диалог создания новой ветви

Создайте новую ветвь с именем test.

Теперь внесите изменение в файл index.html, чтобы перейти в новую ветвь test, например, добавьте текст this is the new test branch.

Сохраните эти изменения ветви test в репозитории. Затем нажмите на имя ветви в левом нижнем углу еще раз, чтобы переключиться обратно на главную ветвь master.

После переключения обратно на ветвь master вы увидите, что текст this is the new test branch, сохраненный для ветви test, отсутствует в главной ветви.

5. Работа с удаленными репозиториями

В этом учебном модуле мы не будем вдаваться в детали, но панель Source Control также предоставляет доступ для работы с удаленными репозиториями. Если вы уже работали с удаленными репозиториями, то вы увидите знакомые вам команды, такие как pull, sync, publish, stash и т. д.

6. Установка полезных расширений

В VS Code имеется не только множество встроенных функций для Git, но и несколько очень популярных расширений, добавляющих дополнительные функции.

Git Blame

Это расширение дает возможность просматривать информацию Git Blame в панели состояния для текущей выделенной строки.

Английское слово Blame имеет значение «винить», но не стоит беспокоиться — расширение Git Blame призвано сделать процесс разработки более практичным, а не обвинять кого-то в чем-то плохом. Идея «винить» кого-то за изменения кода относится не к буквальному возложению вины, а к идентификации человека, к которому следует обращаться с вопросами в отношении определенных частей кода.

Как вы видите на снимке экрана, это расширение выводит на нижней панели инструментов небольшое сообщение, указывающее, кто изменял текущую строку кода, и когда было сделано это изменение.

Git Blame на нижней панели инструментов

Git History

Хотя вы можете просматривать текущие изменения, сравнивать версии и управлять ветвлением с помощью встроенных функций VS Code, они не дают возможности просматривать историю Git. Расширение Git History решает эту проблему.

Как можно увидеть на снимке ниже, это расширение позволяет тщательно изучать историю файла, автора, ветви и т. д. Чтобы активировать показанное ниже окно Git History, нажмите на файл правой кнопкой мыши и выберите пункт Git: View File History:

Результаты работы расширения Git History

Также вы сможете сравнивать ветви и записанные в репозиторий версии, создавать ветви из записанных версий и т. д.

GitLens дополняет возможности Git, встроенные в Visual Studio Code. Это расширение помогает визуализировать принадлежность кода через аннотации Git Blame и линзу кода, просматривать и изучать репозитории Git из среды VS Code, получать полезные аналитические данные с помощью мощных команд сравнения, а также выполнять многие другие задачи.

Расширение Git Lens — одно из самых мощных и популярных среди сообщества разработчиков расширений. В большинстве случаев его функции могут заменить каждое из вышеперечисленных двух расширений.

В правой части текущей строки, над которой вы работаете, отображается небольшое сообщение о том, кто внес изменение, когда это было сделано, а также сообщение о записи изменения в репозиторий. При наведении курсора на это сообщение выводится всплывающий блок с дополнительной информацией, включая само изменение кода, временную метку и т. д.

Функционал Git Blame в Git Lens

Также данное расширение предоставляет много функций, связанных с историей Git. Вы можете легко получить доступ к разнообразной информации, включая историю файлов, сравнение с предыдущими версиями, открытие определенных редакций и т. д. Чтобы открыть эти опции, вы можете нажать на текст на нижней панели состояния, где указан автор, изменивший строку кода, а также время ее изменения.

При этом откроется следующее окно:

Функционал Git History в Git Lens

Это расширение имеет очень много функций, и потребуется время, чтобы разобраться со всеми открываемыми им возможностями.

**Хостинг GitHub Pages**

Что такое GitHub Pages

Git — это самая популярная система контроля версий в мире. GitHub Pages - это бесплатный хостинг для статических файлов.

Если вы создали сайт и оставили его у себя на ПК, то посмотреть его можно только с вашего ПК. Чтобы открыть его с другого устройства, нужно перенести файлы сайта на другое устройство. Чтобы не переносить эти файлы на все устройства в мире, их нужно загружать в облачные хранилища. Облачные хранилища для сайтов называются хостингами.

Что такое хостинг и зачем они нужны сайту

Стандартные хостинги платные. И они скрывают технические файлы сайта от всех, кроме владельцев. GItHub Pages — это бесплатный хостинг с открытой структурой. Это значит, что загрузить туда свои файлы может кто угодно. И любой пользователь может посмотреть все загруженные файлы проекта.

GitHub Pages сильно упрощают поиск работы для начинающих специалистов. Если вы освоите этот инструмент, то он выгодно выделит вас на фоне конкурентов. А HR—менеджеру будет проще и приятнее оценить ваши скиллы.

Как подключить свой домен к проекту на GitHub Page

Перейдите в пункт «Settings», раздел «GitHub Pages». Поменяйте в пункте «Source» значение на «main». Нажмите «Save». Страница обновится, появится поле «Custom Domain». Впишите в него ваш домен.

Теперь нужно зайти в редактор DNS-записей вашего провайдера, создать запись типа А и задать ей значение «192.30.252.153». Пункт к этому месту разный у всех провайдеров, но у них есть неплохие инструкции.

**Первоначальная настройка Git, регистрация на GitHub**

1. Первоначальная настройка Git:

Это нужно сделать только один раз — при обновлении версии Git настройки сохранятся. Но, при необходимости, вы можете поменять их в любой момент, выполнив те же команды снова.

В состав Git входит утилита git config, которая позволяет просматривать и настраивать параметры, контролирующие все аспекты работы Git, а также его внешний вид. Эти параметры могут быть сохранены в трёх местах:

Файл [path]/etc/gitconfig содержит значения, общие для всех пользователей системы и для всех их репозиториев. Если при запуске git config указать параметр --system, то параметры будут читаться и сохраняться именно в этот файл. Так как этот файл является системным, то вам потребуются права суперпользователя для внесения изменений в него.

Файл ~/.gitconfig или ~/.config/git/config хранит настройки конкретного пользователя. Этот файл используется при указании параметра --global и применяется ко всем репозиториям, с которыми вы работаете в текущей системе.

Файл config в каталоге Git (т. е. .git/config) репозитория, который вы используете в данный момент, хранит настройки конкретного репозитория. Вы можете заставить Git читать и писать в этот файл с помощью параметра --local, но на самом деле это значение по умолчанию. Неудивительно, что вам нужно находиться где-то в репозитории Git, чтобы эта опция работала правильно.

Настройки на каждом следующем уровне подменяют настройки из предыдущих уровней, то есть значения в .git/config перекрывают соответствующие значения в [path]/etc/gitconfig.

В системах семейства Windows Git ищет файл .gitconfig в каталоге $HOME (C:\Users\$USER для большинства пользователей). Кроме того, Git ищет файл [path]/etc/gitconfig, но уже относительно корневого каталога MSys, который находится там, куда вы решили установить Git при запуске инсталлятора.

Если вы используете Git для Windows версии 2.х или новее, то так же обрабатывается файл конфигурации уровня системы, который имеет путь C:\Documents and Settings\All Users\Application Data\Git\config в Windows XP или C:\ProgramData\Git\config в Windows Vista и новее. Этот файл может быть изменён только командой git config -f <file>, запущенной с правами администратора.

Чтобы посмотреть все установленные настройки и узнать где именно они заданы, используйте команду:

$ git config --list --show-origin

Имя пользователя - первое, что вам следует сделать после установки Git — указать ваше имя и адрес электронной почты. Это важно, потому что каждый коммит в Git содержит эту информацию, и она включена в коммиты, передаваемые вами, и не может быть далее изменена:

$ git config --global user.name "Имя"

$ git config --global user.email Почта

Опять же, если указана опция --global, то эти настройки достаточно сделать только один раз, поскольку в этом случае Git будет использовать эти данные для всего, что вы делаете в этой системе. Если для каких-то отдельных проектов вы хотите указать другое имя или электронную почту, можно выполнить эту же команду без параметра --global в каталоге с нужным проектом.

Многие GUI-инструменты предлагают сделать это при первом запуске.

Выбор редактора

Теперь, когда вы указали своё имя, самое время выбрать текстовый редактор, который будет использоваться, если будет нужно набрать сообщение в Git. По умолчанию Git использует стандартный редактор вашей системы, которым обычно является Vim. Если вы хотите использовать другой текстовый редактор, например, Emacs, можно проделать следующее:

$ git config --global core.editor emacs

В системе Windows следует указывать полный путь к исполняемому файлу при установке другого текстового редактора по умолчанию. Пути могут отличаться в зависимости от того, как работает инсталлятор.

В случае с Notepad++, популярным редактором, скорее всего вы захотите установить 32-битную версию, так как 64-битная версия ещё не поддерживает все плагины. Если у вас 32-битная Windows или 64-битный редактор с 64-битной системой, то выполните следующее:

$ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"

Примечание

Vim, Emacs и Notepad++ — популярные текстовые редакторы, которые часто используются разработчиками как в Unix-подобных системах, таких как Linux и Mac, так и в Windows. Если вы используете другой редактор или его 32-битную версию, то обратитесь к разделу Команды git config core.editor за дополнительными инструкциями как использовать его совместно с Git.

Предупреждение

В случае, если вы не установили свой редактор и не знакомы с Vim или Emacs, вы можете попасть в затруднительное положение, когда какой-либо из них будет запущен. Например, в Windows может произойти преждевременное прерывание команды Git при попытке вызова редактора.

Настройка ветки по умолчанию

Когда вы инициализируете репозиторий командой git init, Git создаёт ветку с именем master по умолчанию. Начиная с версии 2.28, вы можете задать другое имя для создания ветки по умолчанию.

Например, чтобы установить имя main для вашей ветки по умолчанию, выполните следующую команду:

$ git config --global init.defaultBranch main

Проверка настроек

Если вы хотите проверить используемую конфигурацию, можете использовать команду git config --list, чтобы показать все настройки, которые Git найдёт:

$ git config --list

user.name=John Doe

user.email=johndoe@example.com

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

...

Некоторые ключи (названия) настроек могут отображаться несколько раз, потому что Git читает настройки из разных файлов (например, из /etc/gitconfig и ~/.gitconfig). В таком случае Git использует последнее значение для каждого ключа.

Также вы можете проверить значение конкретного ключа, выполнив git config <key>:

$ git config user.name

John Doe

Примечание

Так как Git читает значение настроек из нескольких файлов, возможна ситуация когда Git использует не то значение что вы ожидали. В таком случае вы можете спросить Git об origin этого значения. Git выведет имя файла, из которого значение для настройки было взято последним:

$ git config --show-origin rerere.autoUpdate

file:/home/johndoe/.gitconfig false

2. Регистрация на GitHub

Чтобы начать работать с GitHub, нужно зарегистрироваться на нём, если вы ещё этого не сделали. Регистрация предельно проста:

* Переходим на сайт GitHub.
* Нажимаем кнопку Sign up (зарегистрироваться).
* На странице регистрации вам предложат ввести обязательные данные:
* имя пользователя;
* адрес электронной почты;
* пароль.

После этого вы попадёте на небольшой опрос от GitHub, который можете просто пропустить, нажав skip this step.

**Удаление репозитория**

Удаление локального Git-репозитория. Если с созданием и клонированием репозиториев все понятно, то как быть, когда папка с проектом уже существует, но в нее нужно поместить новые данные, удалив предыдущие, или когда Git-репозиторий на сервере больше не нужен. В таком случае осуществляется стандартное удаление.

В корне каталога с проектом необходимо избавиться от папки .git, о которой уже шла речь выше. Так вы удаляете только ту информацию, которая связана с Git, но сам проект остается. В Linux через Терминал вы должны перейти к каталогу с проектом и ввести следующую команду:

rm -rf .git

Еще один вариант – удаление .gitignore и .gitmodules в случае их наличия. Тогда команда меняет свой вид на:

rm -rf .git\*

Остается только убедиться в отсутствии скрытой папки, которая может помешать инсталляции новой. Для этого существует простая команда ls -lah, выполнить которую необходимо с указанием того же каталога.

**Создание файла gitignore**

Использование gitignore для предотвращения отслеживания файлов

Создайте gitignore -файл в репозитории Git, чтобы не допустить размещения в Git нежелательных файлов. Share . gitignore в ветви по умолчанию в репозитории. Вы и ваша команда можете обновить файл, чтобы изменить типы файлов, которые следует игнорировать.

Создать gitignore из командной строки. Перейдите в репозиторий Git и выполните одну из следующих команд, используя сведения о репозитории:

* Windows: fsutil file createnew C:\Users\frank\myrepo\.gitignore 0
* Linux и macOS: touch /home/frank/myrepo/.gitignore

Git применяет . gitignore к папке и вложенным папкам, где она находится. Мы рекомендуем разместить . gitignore в корневой папке репозитория, чтобы избежать путаницы.

Настройка. Gitignore. Измените gitignore , чтобы включить в репозиторий типы файлов, пути и шаблоны файлов. Git начинает пропускать эти файлы сразу после обновления . gitignore. Если другим пользователям вашей команды требуется один и тот же набор игнорируемых файлов, обязательно зафиксируйте изменения.

Используйте текстовый редактор, например следующий пример, в котором используется vim:

vim /home/frank/myrepo/.gitignore

Каждая строка в . gitignore исключает файл или набор файлов, соответствующих шаблону. Полный синтаксис gitignore очень гибок. Ниже приведены некоторые примеры наиболее распространенных записей.

# ignore a single file

mycode.class

# ignore an entire directory

/mydebugdir/

# ignore a file type

\*.json

Windows users: все пути к файлам в gitignore -файле используют разделитель косой черты, а не обратную косую черту.

Пропускать файлы только в вашей системе

Ваш gitignore является общим для участников команды, так как файл зафиксирован и отправлен в репозиторий Git. Чтобы исключить файлы только в системе, измените файл . git/info/Exclude в локальном репозитории. Изменения в этом файле не являются общими для других пользователей. Они применяются только к файлам в этом репозитории. Для этого файла используется тот же синтаксис , что и в файле. gitignore.

Пропускать файлы во всех репозиториев в системе

Настройте Global . gitignore для использования во всех репозиториев в системе с помощью программы командной строки , как показано в следующем примере:

git config core.excludesfile C:\Users\frank\.gitignore\_global

Этот подход удобен для игнорирования целых типов файлов, которые вы не хотите зафиксировать, например скомпилированных двоичных файлов.

Игнорировать изменения в зафиксированных файлах

Временно игнорировать изменения

Во время разработки удобно отключить отслеживание изменений файлов в файле, зафиксированном в репозитории Git. Этот подход удобен при настройке параметров или файлов конфигурации, которые являются частью источника проекта, для собственной рабочей среды.

git update-index --skip-worktree <file>

Возобновление отслеживания файлов с помощью следующей команды:

git update-index --no-skip-worktree <file>

Вместо этого можно использовать следующие параметры. Эти параметры предназначены главным образом для отметки файлов, которые не должны изменяться разработчиками.

Чтобы отключить отслеживание изменений, выполните следующие действия.

git update-index --assume-unchanged <file>

Чтобы возобновить отслеживание изменений, выполните следующие действия.

git update-index --no-assume-unchanged <file>

Безвозвратная отмена отслеживания файла

Если файл уже записан в Git, он .gitignore не применяется. Git продолжит отслеживание изменений в этом файле.

Если вы хотите отключить отслеживание файла, необходимо явно указать git, который он должен удалить из отслеживания. Следуя этим указаниям, файл останется в локальном рабочем каталоге, но не будет относиться к записи в Git.

Добавьте файл в .gitignore

Выполните следующую команду: git rm --cached <file>

Зафиксируйте удаление файла и обновленный файл . gitignore в репозитории.

**Создание файла README.md синтаксис Markdown**

Для описания проектов на GitHub используется README.md, который пишется на языке разметки markdown. Что и как поддерживается расписано ниже. Также существует еще один формат - reStructuredText, описание которого вынесено в отдельный файл README.rst.

Заголовки задаются значками решётки - сколько их, такого уровня заголовок. Как и в HTML возможен уровень от 1 до 6.

# h1

## h2

### h3

Списки

Маркированный

Задать маркированный список можно несколькими символами:

- ul

+ ul

+ ul - для вложенных списков нужно поставить 2 пробела

+ ul - третий уровень

+ ul - четвёртый уровень

+ ul - пятый и т.д.

\* ul

\* ul

\* ul - если ставить по одному пробелу, то выше второго уровня не подняться

Нумерованным спискам безразлично с какого числа вы начнёте его - он начнёт с единицы. Точка после числа обязательна. Работа с вложенностями аналогична маркированному списку.

1. ol

1. ol

1. ol

1. ol

2. ol

2. ol - всё равно 3

Если же Вам нужно число с точкой, но не нужен список, то можно сделать так:

1961\. Текст

Нумерованный и маркированный списки можно смешивать:

\* ul

1. ol

1. ol

1. ol

\* ul

\* ul

Либо просто вставить ссылку, либо дополнительно задать текст ссылки (пробела между скобками быть не должно):

http://example.com

[example.com](http://example.com)

Можно выводить через "константу", которую можно задать в любом месте, обычно их задают в конце документа.

[example site]

[example site]:http://example.com

Цитаты

> Цитата

> > Вложенная цитата

> Продолжение цитаты

Начертание шрифта

\*\*bold\*\*

\*italic\*

\*\*\*bold italic\*\*\*

\_italic too\_

\*\*Можно использовать \_курсив\_ внутри жирного текста.\*\*

Вставки кода. Если нужно выделить слово или фразу внутри строки, то используются одинарные обратные апострофы (`), для выделения в блоки - тройные. Дополнительно можно задавать язык кода внутри блока:

Можно использовать `light-code` внутри строки

```

code block

```

```php

//php code

$foo = new BarClass();

```

Изображения. Описание крайне похоже на описание ссылки, за исключением лишь восклицательного знака в начале.

![alt text](ссылка)

**Откат к коммиту, расширение Git Graph для VS Code**

1. Посмотреть состояние на определенном моменте. Выбираем нужный коммит, и в консоли (и сразу откатываемся на нем): git checkout ID\_коммита

Чтобы посмотреть, состояние проекта на определенном моменте, для этого есть команде git log. Один из способов просмотреть лог (последние 10 коммитов): git log --oneline -10

Теперь чтобы вернуться на последний актуальный коммит:

git checkout master

Чтобы переключиться на определенный коммит, и создать отдельно ветку — набираем в консоли:

git checkout -b имя\_новой\_ветки ID\_коммита

2. Теперь рассмотрим случай, когда возвращаться не нужно, а требуется только удалить все свои незакомиченные изменения (которые ещё не индексировались командой git add): git stash save --keep-index

при этом появится ещё одно состояние проекта (карман).

Удалить все индексированные и неиндексированные изменения:

git stash save

Вернуть обратно: git stash apply

Если «карман» не нужен, его можно удалить: git stash drop

И наконец, если необходимо переключиться на предыдущий коммит, отменить все последующие и продолжить разработку с него, лучший способ – это использовать git revert.

Эта команда создает новые коммиты на каждый последующий от текущего состояния.

Синтаксис следующий:

git revert IDs\_коммитов или git revert ID\_коммита..ID\_коммита

Замет нужно всё это дело подтвердить коммитом:

git commit -m 'что и зачем'

**Добавление описания профиля на GitHub с помощью README.md файла**

Создаем README.md профиля. Для того чтобы создать README-файл, который будет отображаться в профиле, необходимо создать новый публичный репозиторий с названием, полностью повторяющим никнейм пользователя. Также важно не забыть поставить галочку в поле, предлагающем вместе с репозиторием создать и README.md. Во время настройки репозитория нас поздравит Октокот и сообщит о том, что только что мы создали особенный репозиторий и его содержимое будет отображаться на странице профиля. GitHub автоматически сделает базовый шаблон, в который можно вписать информацию о себе, а можно не вписывать и сделать что-то чуть более интересное.

Для редактирования файла понадобятся знание Markdown-разметки и базовое понимание HTML. Оба навыка не сложны в освоении и интуитивно понятны в том объеме, который необходим для кастомизации профиля. На самом деле можно ограничиться только Markdown, но его особенность в том, что текст автоматически выравнивается по левому краю и нет никакой возможности повлиять на это. А HTML в тандеме с Markdown позволяет контролировать расположение объектов на экране. Для создания интерактивных блоков понадобится понимание того, как устроен сервис GitHub Actions, но нужны тоже только лишь базовые знания. Глубокие понадобятся только в том случае, если захочется создать собственный динамический виджет.

**3 Разработка базы данных**

**3.1 Моделирование данных. ERD**

**3.2 Создание базы данных по средствам запросов**

**3.3 Создание аналитических запросов**

**4 Разработка приложения для работы в день С-1 (руководство пользователя)**

**4.1 База данных**

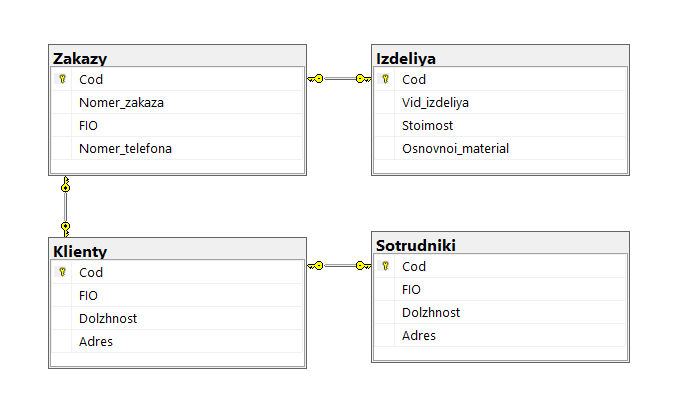
****

Рисунок 30. Схема базы данных

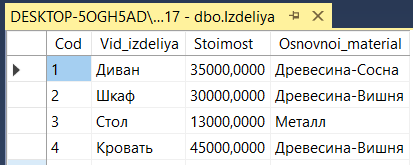
****

Рисунок 30. Таблица Изделия

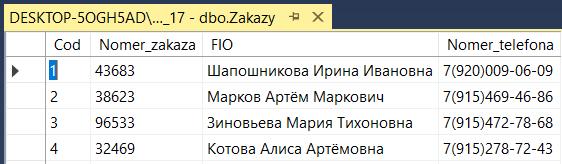
****

Рисунок 30. Таблица Заказы

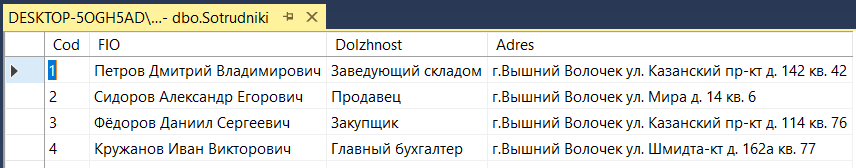
****

Рисунок 30. Таблица Сотрудники

**4.2 Разработка форм авторизации и поиска**

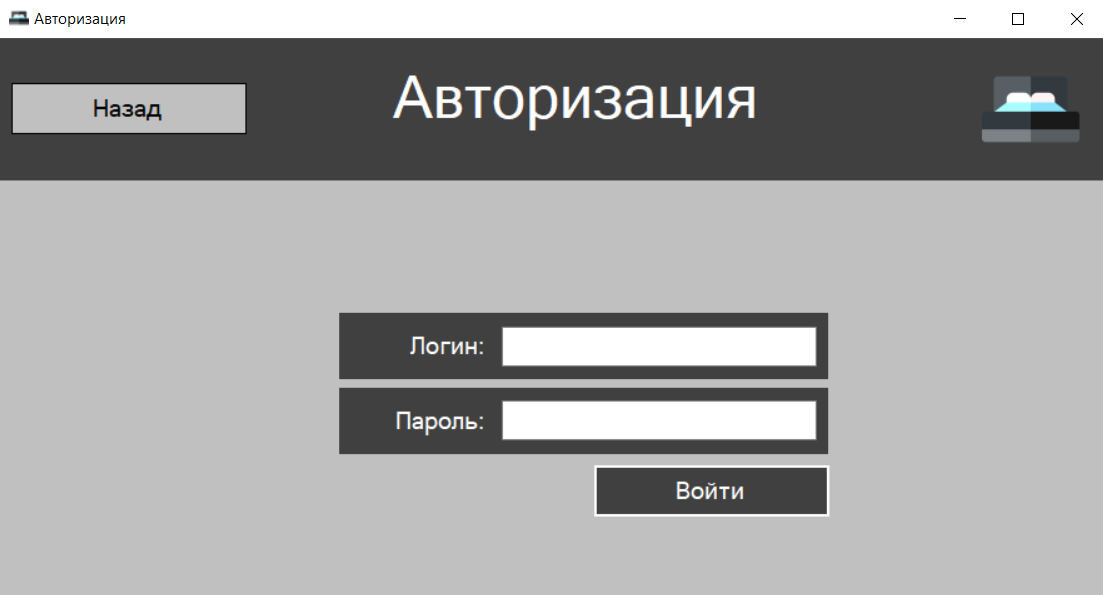
****

Рисунок 30. Форма Авторизации

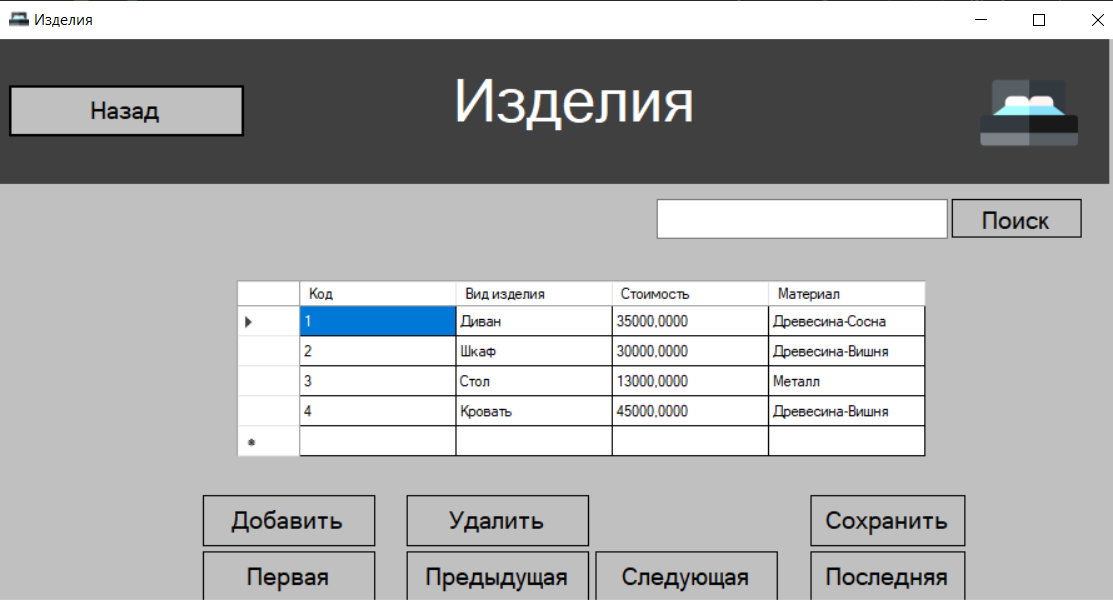
****

Рисунок 30. Форма Изделия Пользователя

**4.3 Разработка форм фильтрации и сортировки**

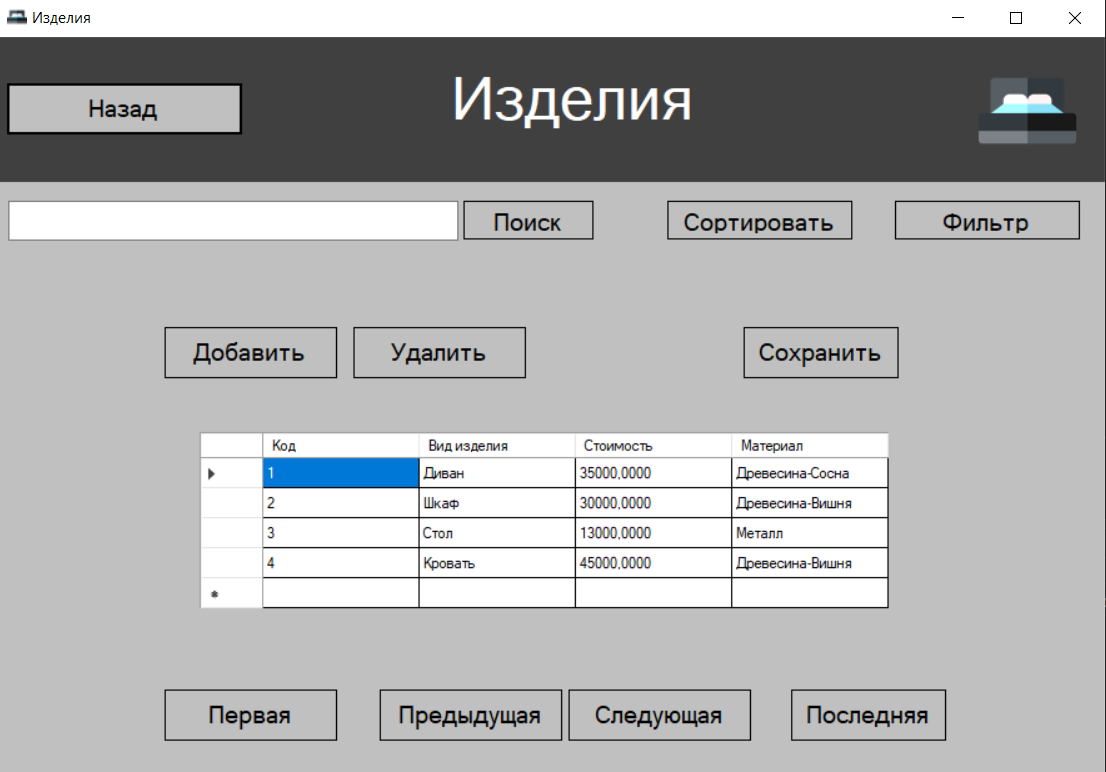
****

Рисунок 30. Форма Изделия Администратора

**5 Разработка desktop приложений**

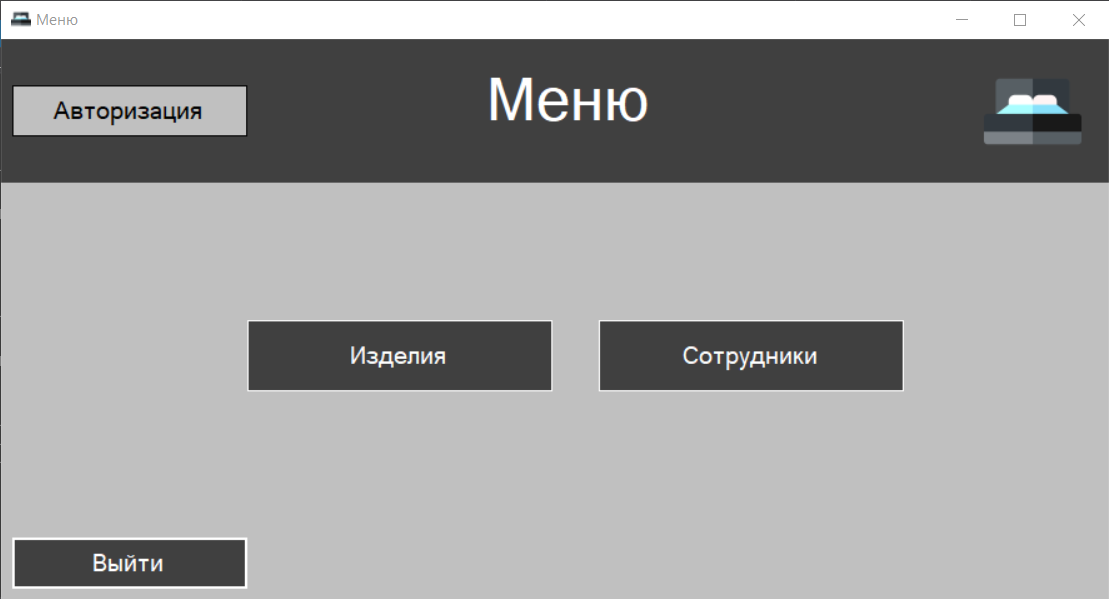
****

Рисунок 30. Форма Меню Пользователя

****

Рисунок 30. Форма Меню Администратора

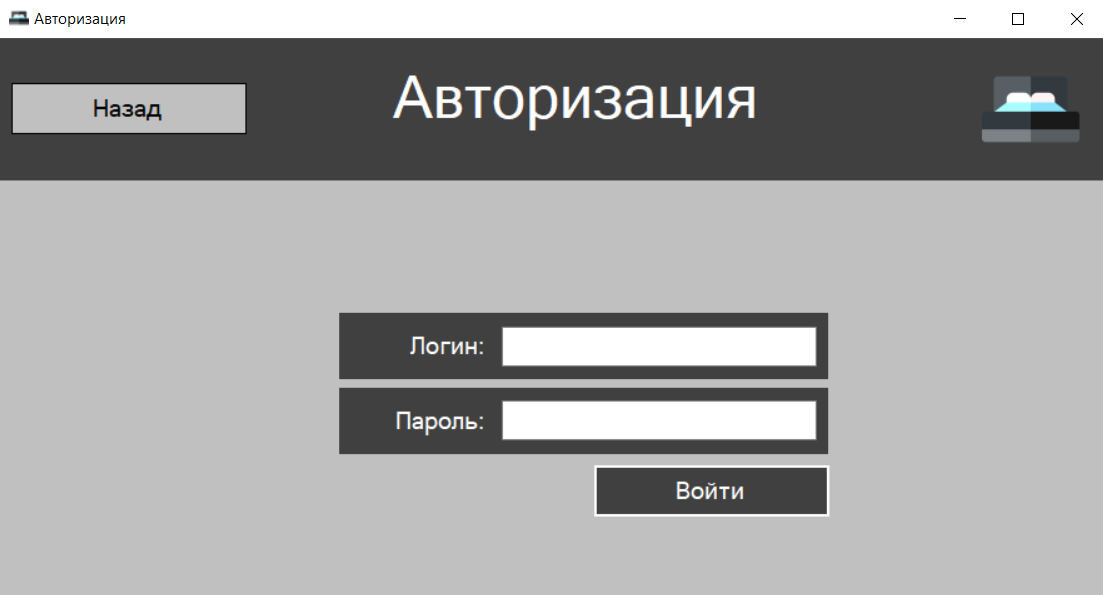
****

Рисунок 30. Форма Авторизации

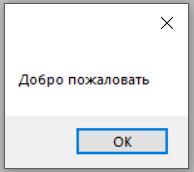
****

Рисунок 30. Диалоговое окно

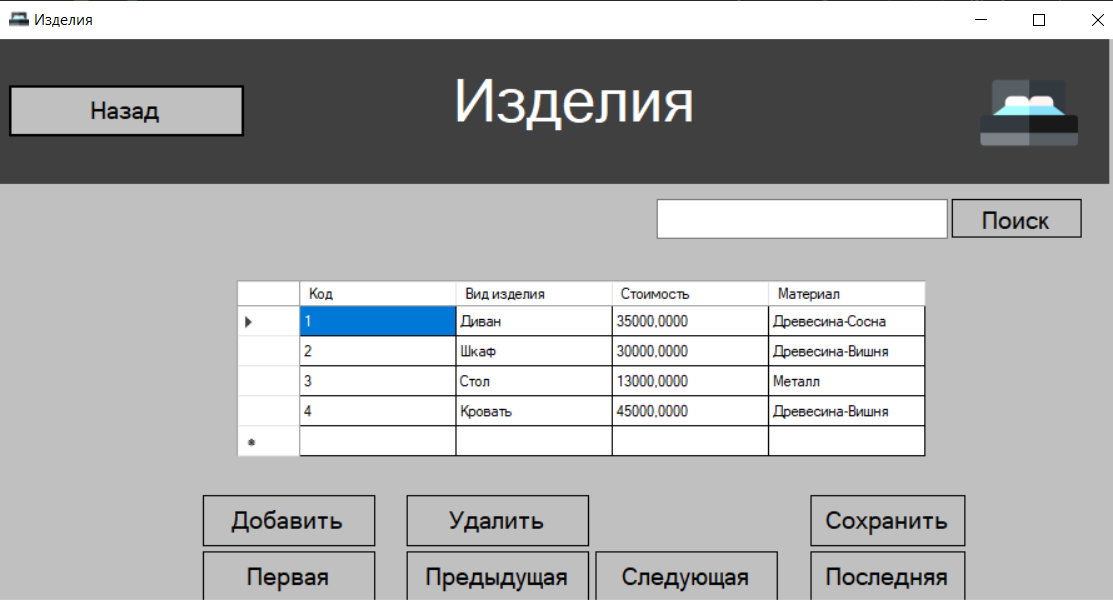
****

Рисунок 30. Форма Изделия Пользователя

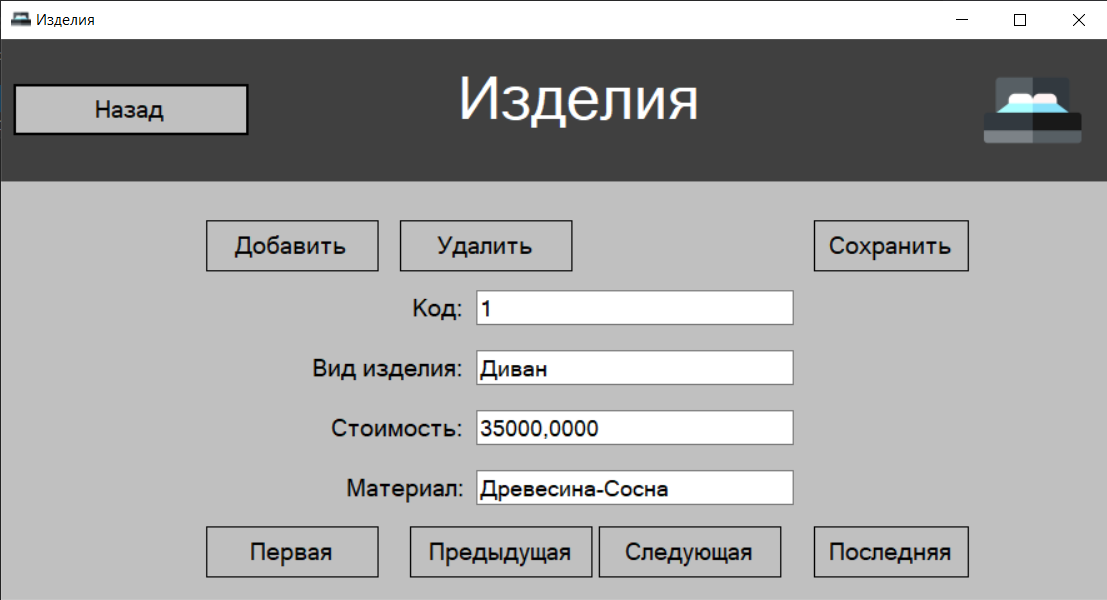
****

Рисунок 30. Форма Изделия Администратора

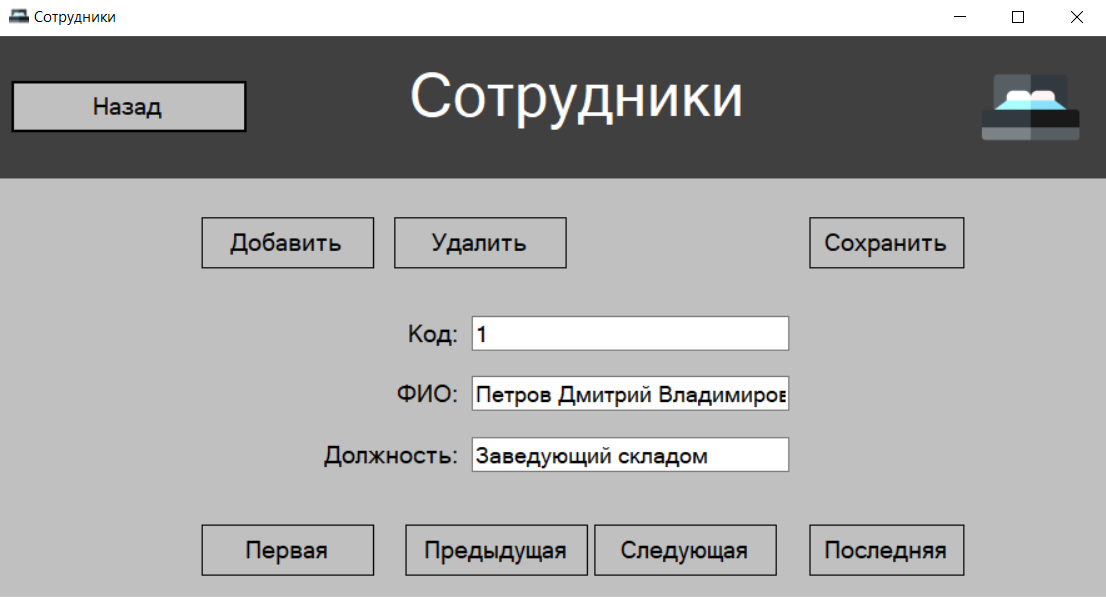
****

Рисунок 30. Форма Сотрудники Пользователя

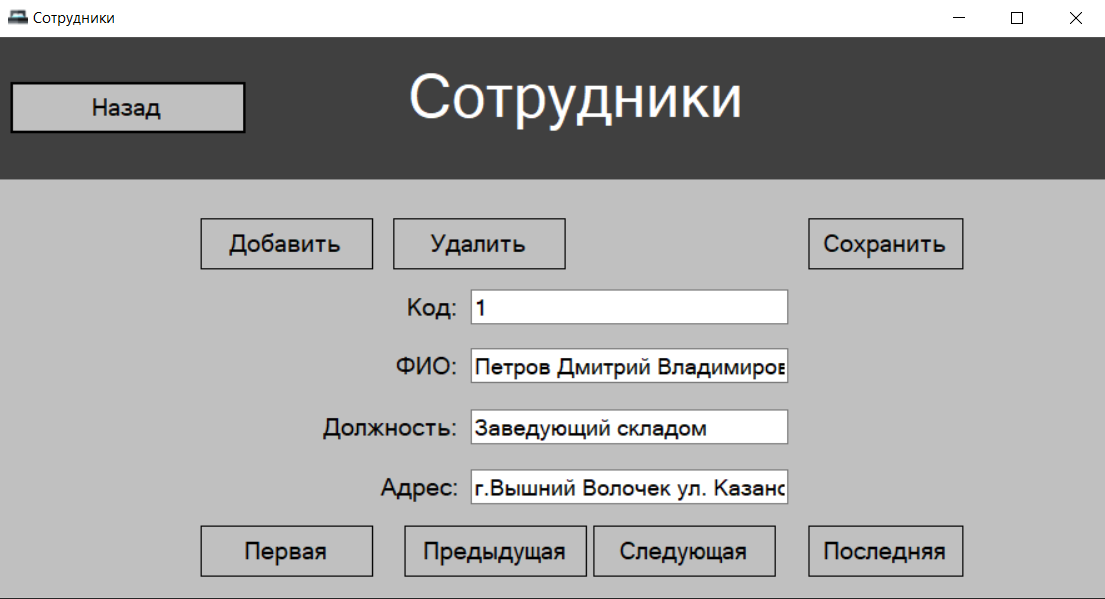
****

Рисунок 30. Форма Сотрудники Администратора

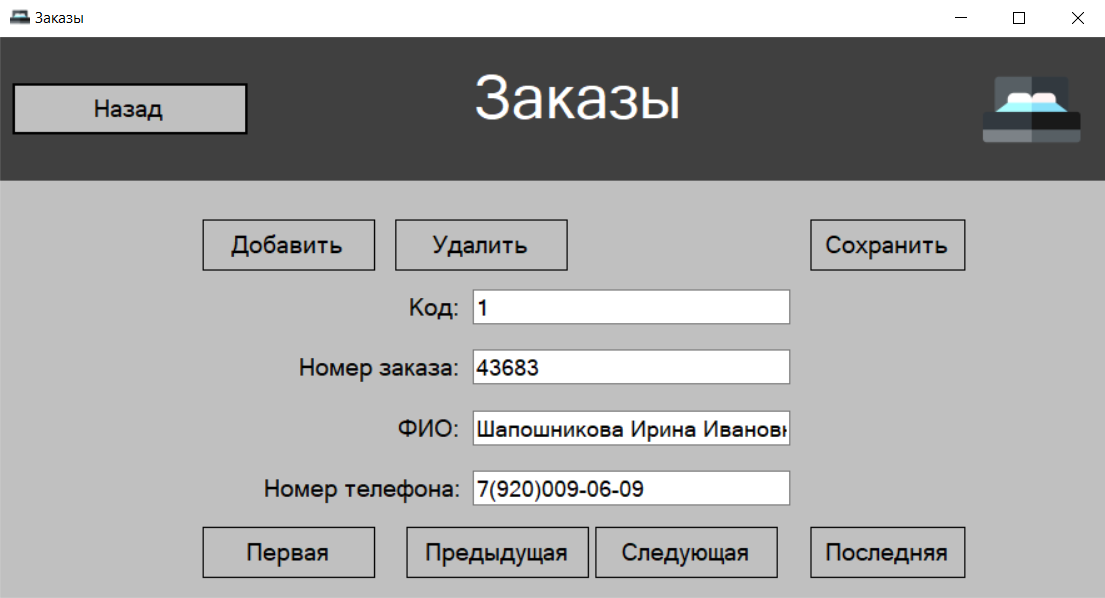
****

Рисунок 30. Форма Заказы Администратора

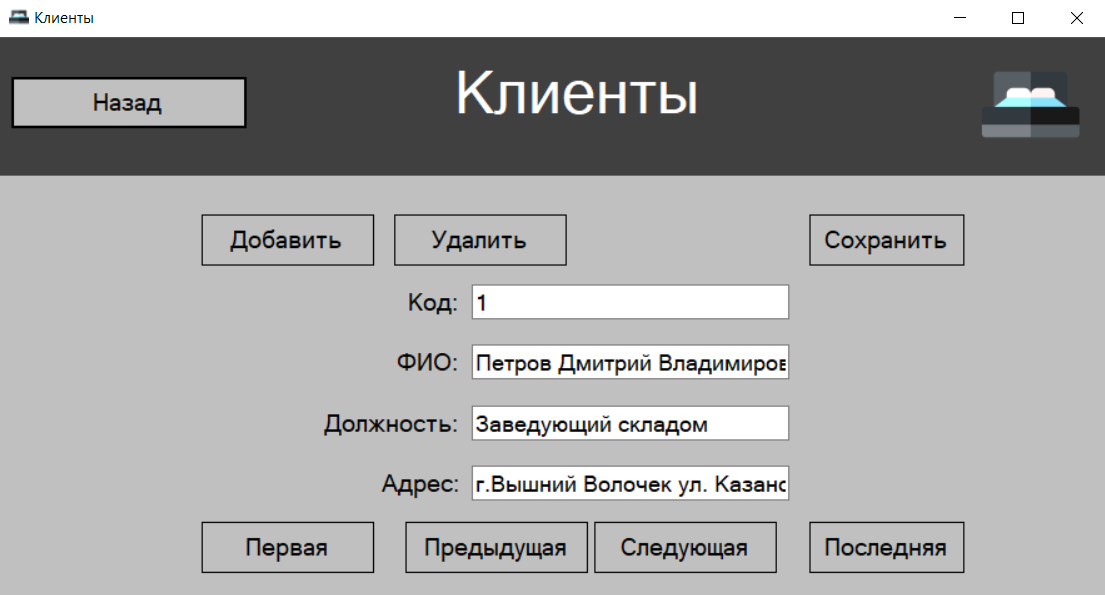
****

Рисунок 30. Форма Клиенты Администратора

**5.2 Разработка ERD диаграмм**

**5.2.1 Работа в MS Visio**

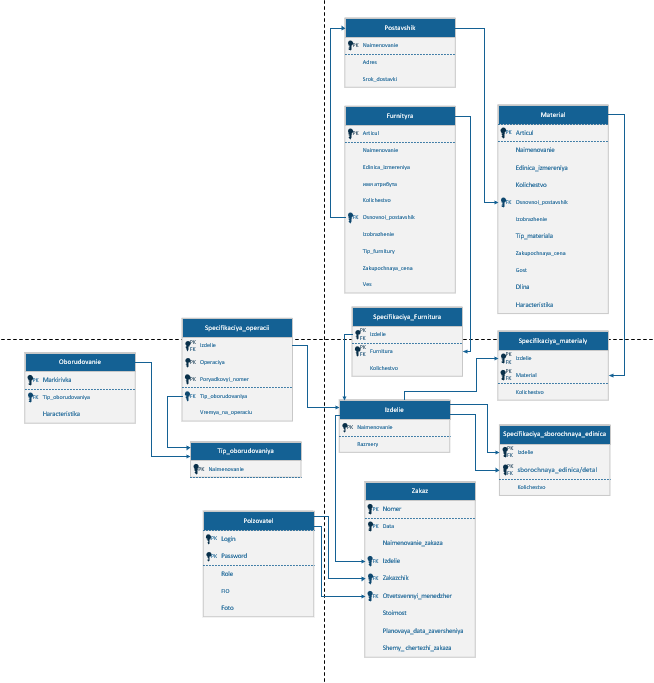
****

Рисунок 30. ERD Диаграмма

**5.2.2 Подготовка данных для импорта**

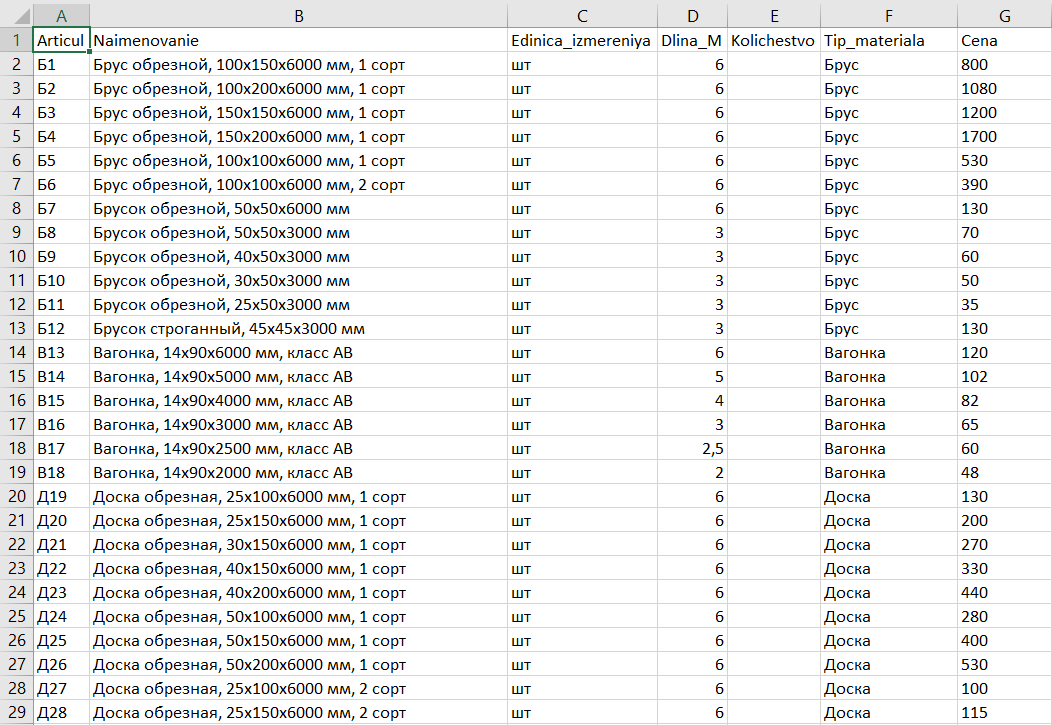
****

Рисунок 30. Таблица Материалы

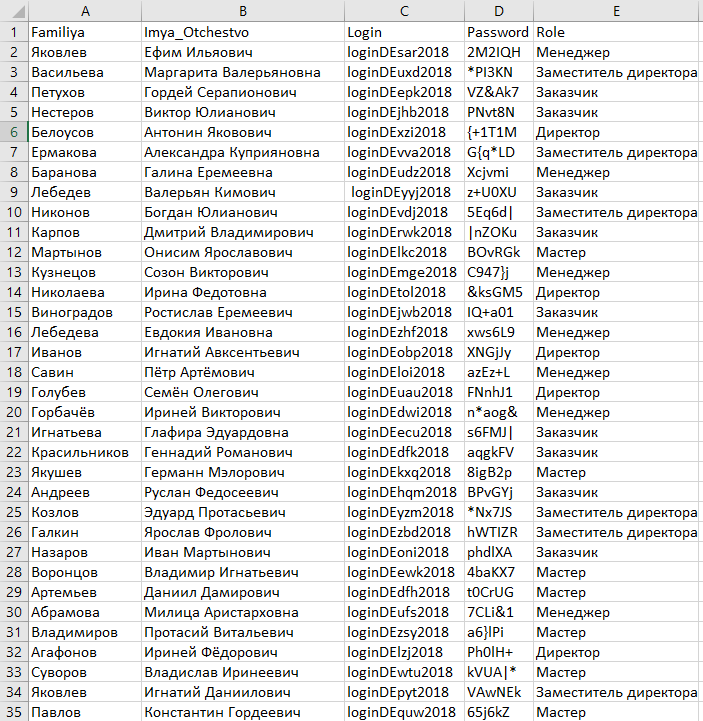
****

Рисунок 30. Таблица Пользователи

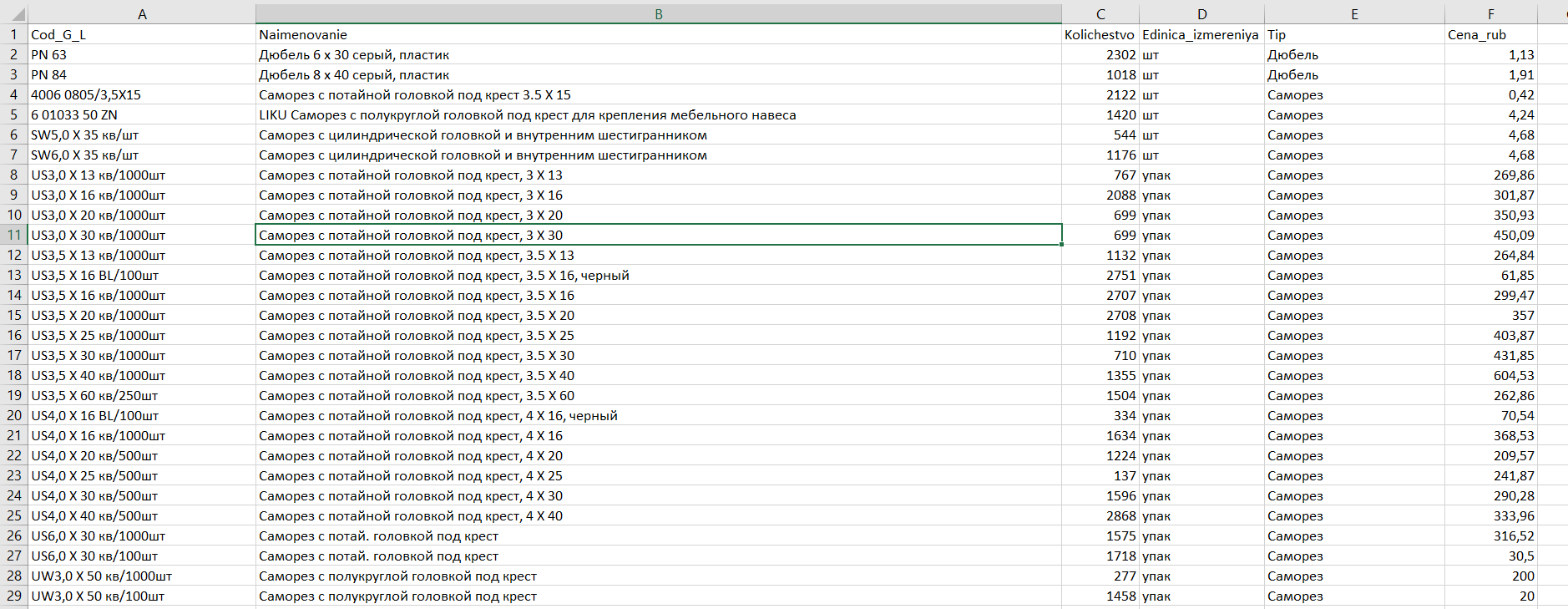
****

Рисунок 30. Таблица Фурнитура

**5.2.3 Разработка базы данных «Мебельная фабрика»**

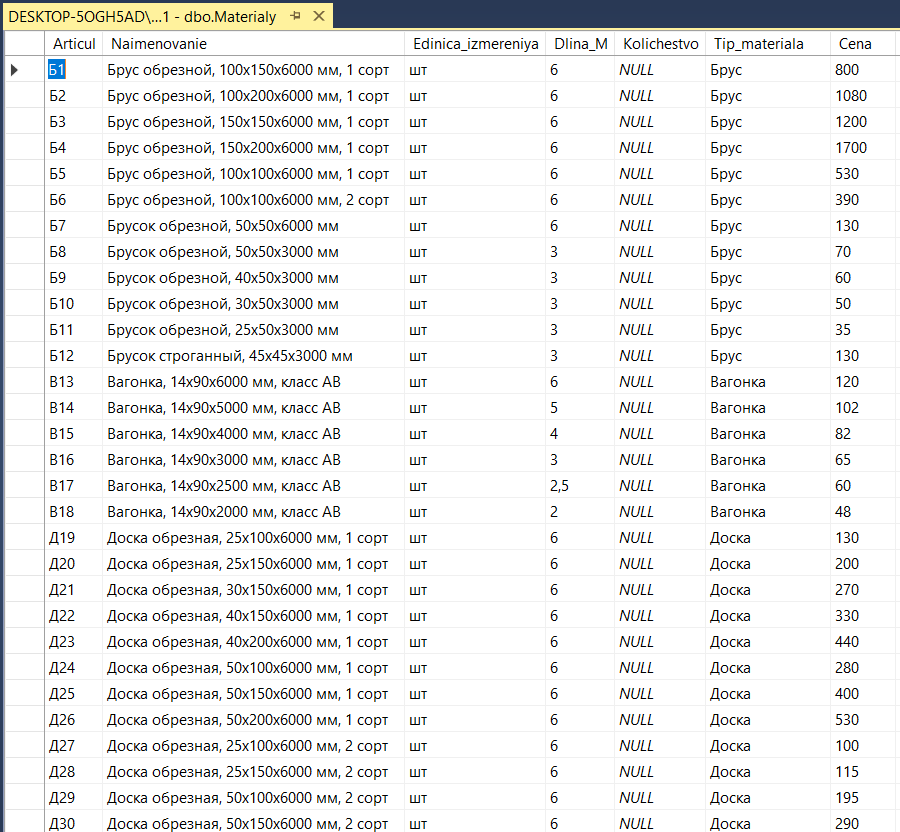
****

Рисунок 30. Таблица Материалы

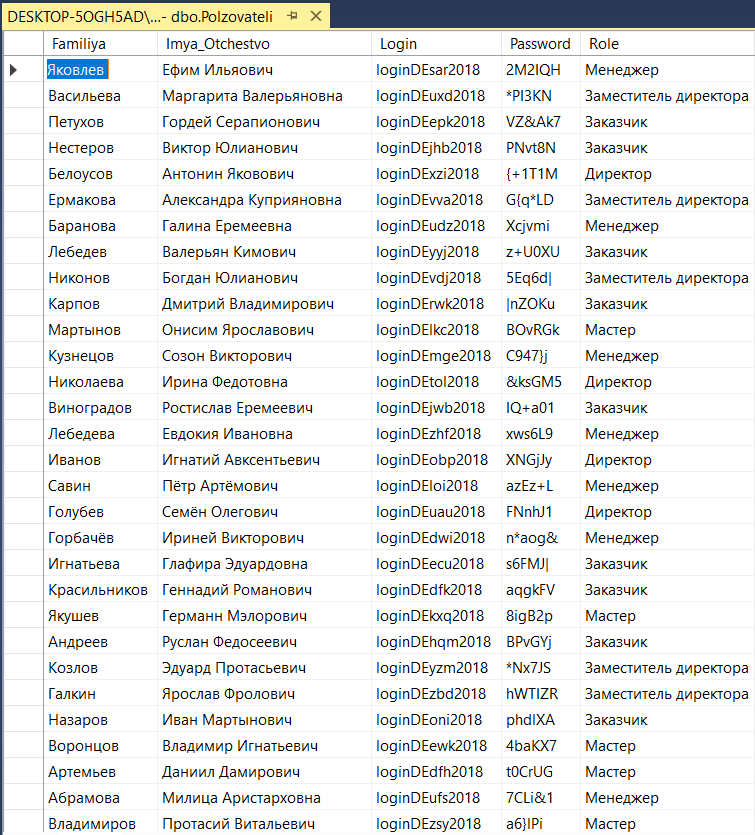
****

Рисунок 30. Таблица Пользователи

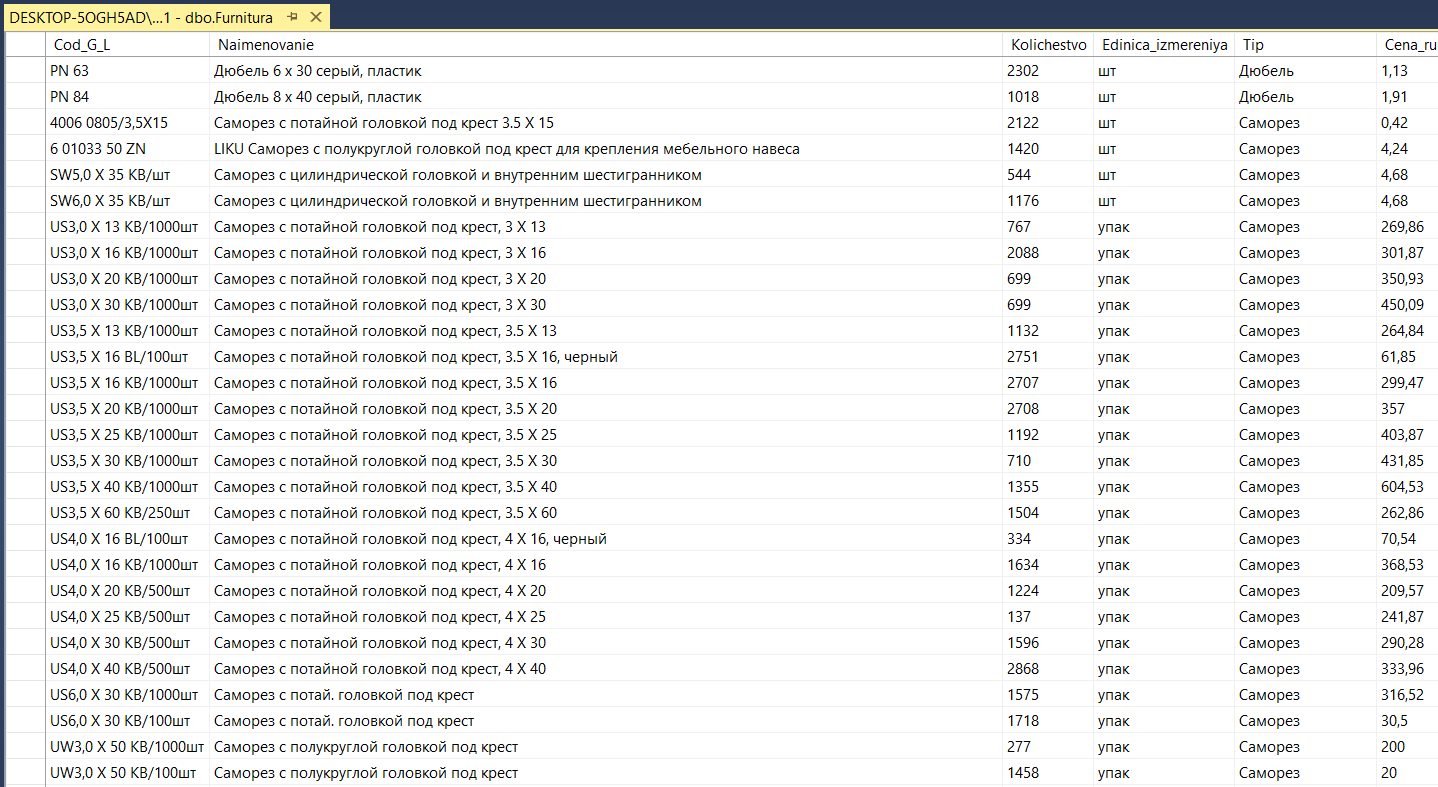
****

Рисунок 30. Таблица Фурнитура

**5.3 Создание приложения**

**5.4 Разработка формы авторизации**