Министерство науки и высшего образования Российской Федерации

федеральное государственное бюджетное образовательное учреждение высшего образования

«Российский экономический университет имени Г.В. Плеханова»

**МОСКОВСКИЙ ПРИБОРОСТРОИТЕЛЬНЫЙ ТЕХНИКУМ**

специальность 09.02.07 «Информационные системы и программирование»

Квалификация: Программист

ПРАКТИЧЕСКИЕ РАБОТЫ

ПО МДК 04.02 «Обеспечение качества функционирования компьютерных систем»

|  |  |
| --- | --- |
| Выполнил студент  группы П50-7-20  Огурцов Антон Алексеевич | Проверил преподаватель  \_\_\_\_\_\_\_\_\_\_\_ К.С. Образцова  «\_\_\_» \_\_\_\_\_\_\_\_\_ 2023 года |

Москва 2023

Оглавление

[Практическая работа №1. 3](#_Toc153057480)

[Практическая работа №2. 9](#_Toc153057481)

[Практическая работа №3. 18](#_Toc153057482)

[Практическая работа №4. 29](#_Toc153057483)

[Практическая работа №5. 38](#_Toc153057484)

[Практическая работа №6. 39](#_Toc153057485)

[Практическая работа №7. 43](#_Toc153057486)

# Практическая работа №1.

Цель работы: Продемонстрировать конфигурацию Git. Создать локальный репозиторий. Продемонстрировать работу основных команд для работы с файлами в репозитории. Создание нескольких различных веток. Описать работу с указателями HEAD.

Управление правами доступа: В централизованной системе управление правами доступа к репозиторию обычно происходит на уровне центрального сервера. В распределенной системе каждый пользователь может выполнять операции с собственной копией репозитория, что позволяет более гибко управлять правами доступа.

Скорость работы: Распределенная система обычно быстрее выполняет операции, так как все данные хранятся локально у каждого пользователя. В централизованной системе операции происходят через удаленное соединение, что может быть медленнее при большом количестве пользователей.

Система контроля версий со списком изменений (например, Subversion) хранят только разницу (delta) между последующими версиями файлов. Когда вы делаете коммит, система сохраняет только изменения, сделанные в файлах, относительно предыдущего коммита, в виде списка изменений. На самом деле, система часто хранит несколько версий файла, чтобы быстрее обрабатывать запросы, основанные на списке изменений. При переключении на предыдущие версии проекта, СКВ применяет все изменения последовательно, чтобы восстановить запрошенную версию проекта.

Система контроля версий — это специальное программное обеспечение, которое используется для управления изменениями в файловой системе, отслеживания и контроля версий документов или кода программы. Она позволяет разработчикам и команде проекта работать над файлами вместе, отслеживать изменения, управлять их версиями и восстанавливаться к предыдущим состояниям проекта.

Коммит (commit) представляет собой операцию, при которой изменения в файловой системе, внесенные программистом, сохраняются в репозитории проекта. Коммит фиксирует изменения в файле или наборе файлов, а также добавляет комментарии или описания к этим изменениям.

Каждый коммит имеет уникальный идентификатор, который позволяет отслеживать историю изменений и восстанавливать предыдущие версии файлов. Коммиты также позволяют работать с несколькими программистами над одним проектом, объединяя изменения, внесенные каждым из них.

Git -- это распределенная система управления версиями, которая используется для отслеживания изменений в коде программного обеспечения. Git позволяет разработчикам работать над проектом одновременно и совместно, сохраняя историю изменений и легко управляя версиями. Git также обеспечивает возможность ветвления и слияния кода, а также гибкую работу с удаленными репозиториями.

Преимущества внедрения DevOps в разработку:

- Более быстрая доставка нового функционала. DevOps способствует автоматизации процесса развертывания и доставки нового функционала пользователям. Благодаря этому, команда разработки может выпускать обновления и новые версии приложения значительно быстрее и с меньшими затратами.

- Более гибкий и масштабируемый процесс разработки. DevOps позволяет легко внедрять изменения и масштабировать процесс разработки в соответствии с потребностями пользователей или рынка. Это позволяет более гибко реагировать на изменения условий и быстро адаптироваться к новым требованиям.

- Снижение затрат на разработку и эксплуатацию. DevOps позволяет оптимизировать процессы разработки и эксплуатации, что сокращает затраты на труд и ресурсы. Это позволяет сделать разработку ПО более эффективной и экономичной.

- Более высокая удовлетворенность пользователей. Благодаря улучшенному качеству ПО, более быстрой доставке нового функционала и более оперативной поддержке, пользователи получают более удовлетворительный опыт использования приложения. Это повышает удовлетворенность пользователей и способствует удержанию и привлечению новых клиентов.

Создание репозитория через команду git init.

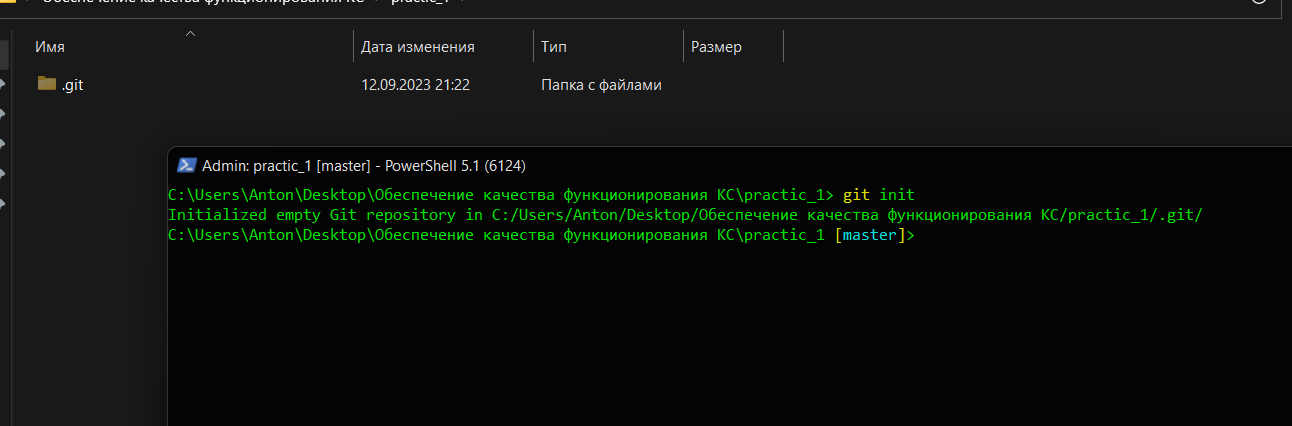


Рисунок 1. git init.

Git config – команда настройки. Для каждого репозитория можно настроить свои настройки или можно использовать –global.

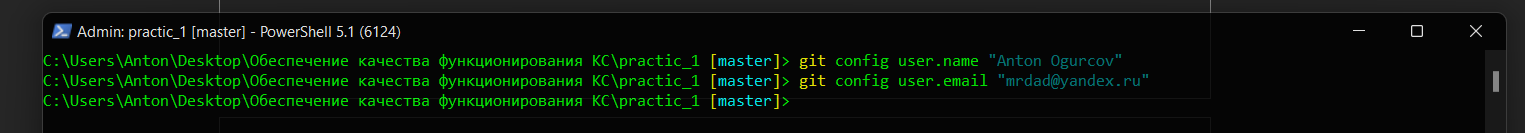


Рисунок 2. git config – настройка пользователя.

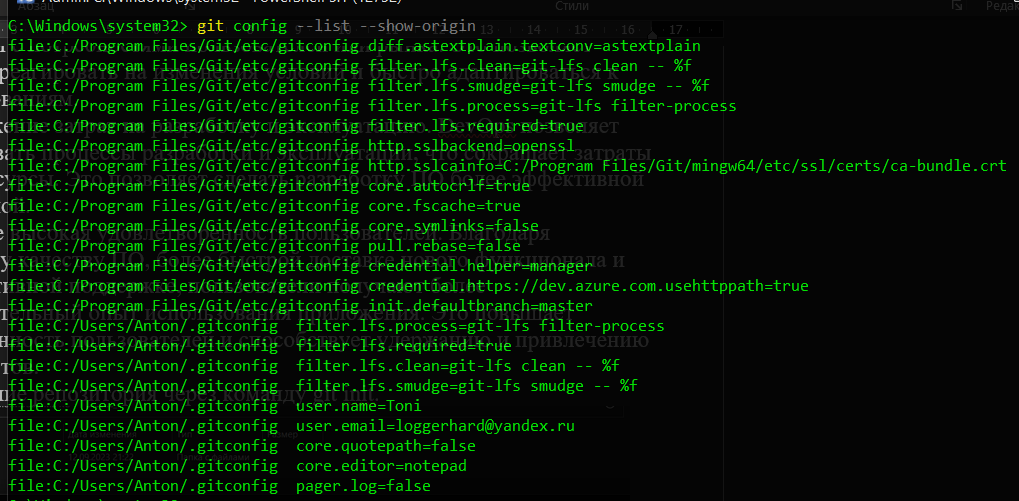


Рисунок 3. git config - список настроек.

При появлении в репозитории нового файла в консоли это отобразиться. И написав команду git add мы будем «следить» за файлом.

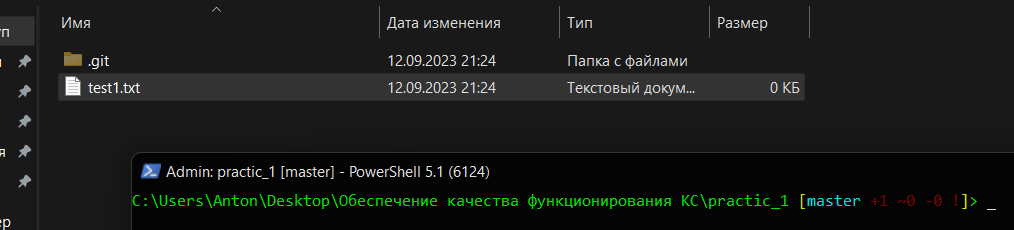


Рисунок 4. Создание любого файла.

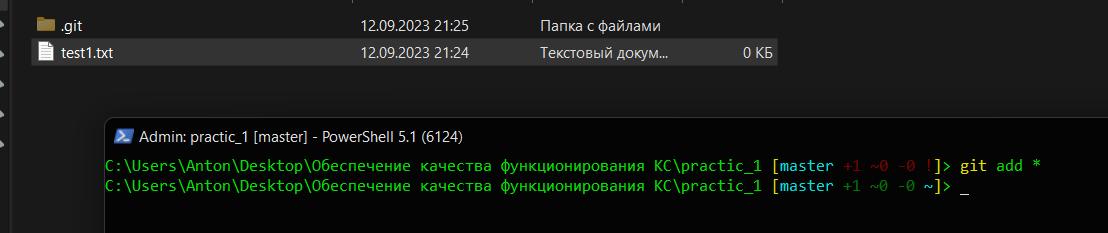


Рисунок 5. git add.

С помощью команды git rm -cached «имя файла» мы перестаём следить за ним. Следовательно, при команде git push файл не отправляется на удалённый репозиторий.

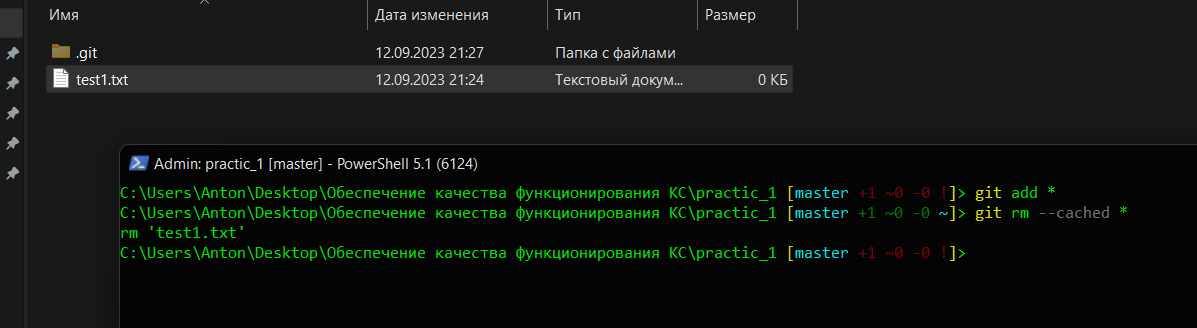


Рисунок 6. git rm --cached.

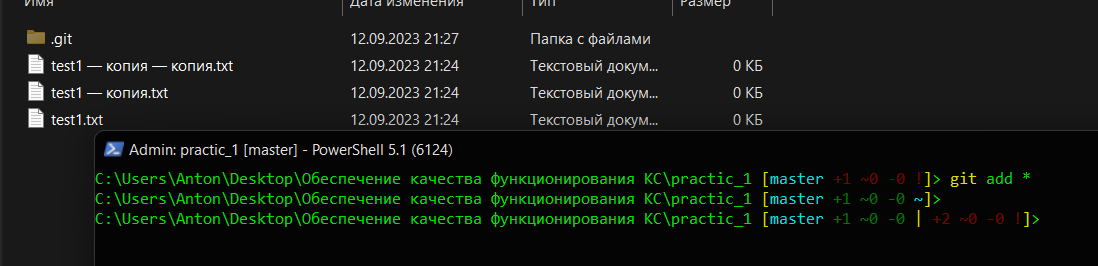


Рисунок 7. git rm - последствия.

Git status – команда, показывающая состояние рабочего репозитория и раздел проиндексированных файлов. Индефиксирование файлов происходит из-за команды git commit.

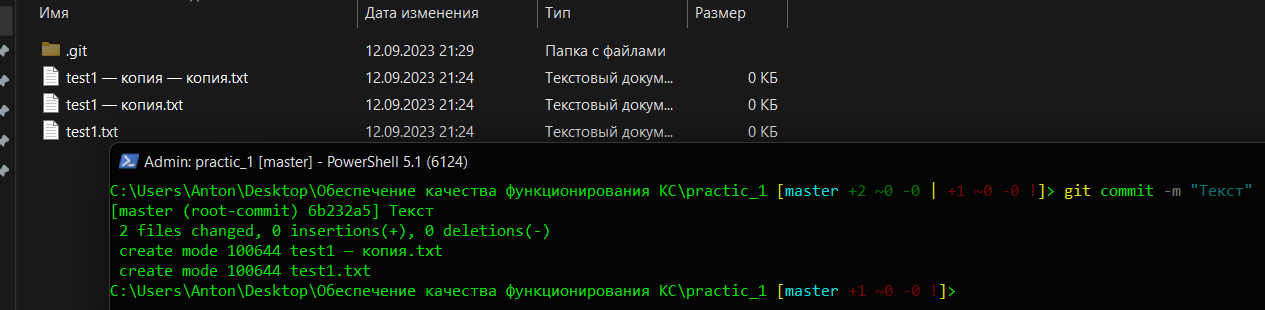


Рисунок 8. git commit.

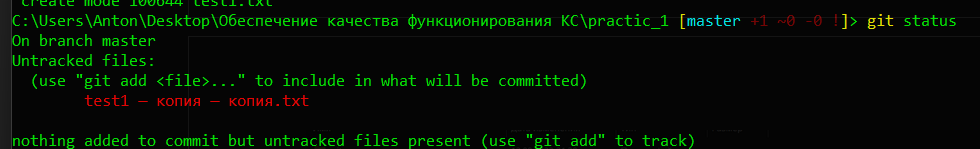


Рисунок 9. git status.

Команда git log – показывает разнообразную команду о коммитах.

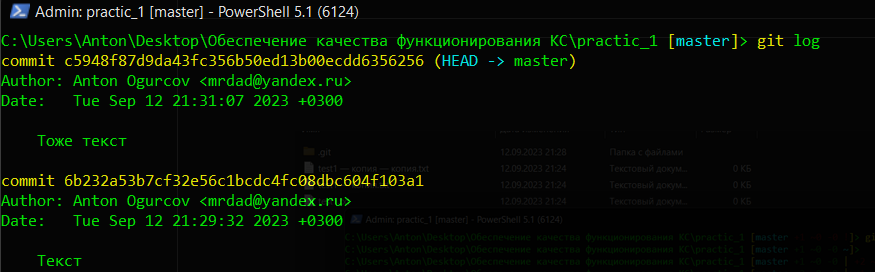


Рисунок 10. git log .

Git switch – команда перехода с ветки на ветку, при дополнении –create создаёт и переходит на новую ветку. Git branch – показывает существующие ветки. При приставке –delete – удаляет ветку.

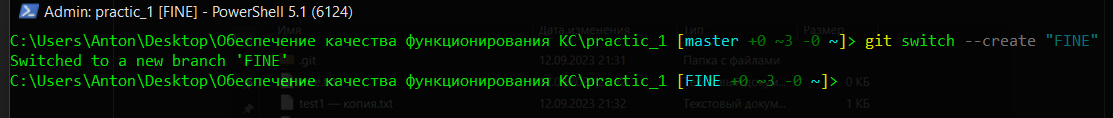


Рисунок 11. git switch --create.

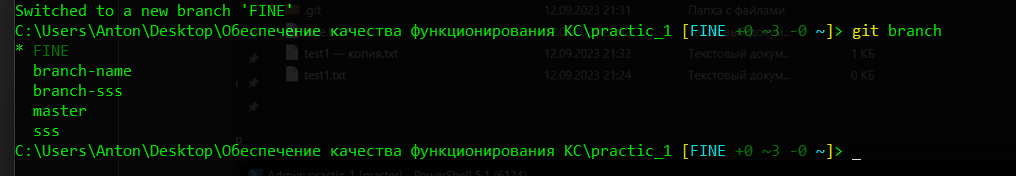


Рисунок 12. git branch.

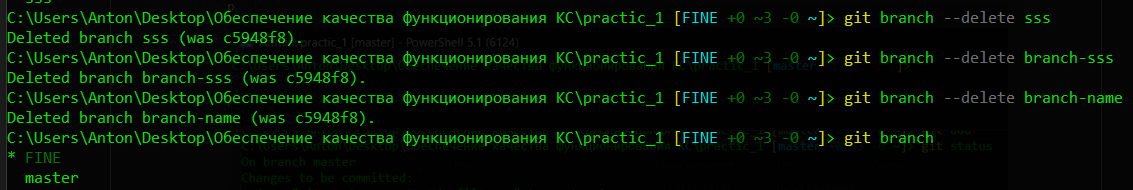


Рисунок 13. git branch --delete.

Git clean – команда для работы с не отлеживающими файлами.

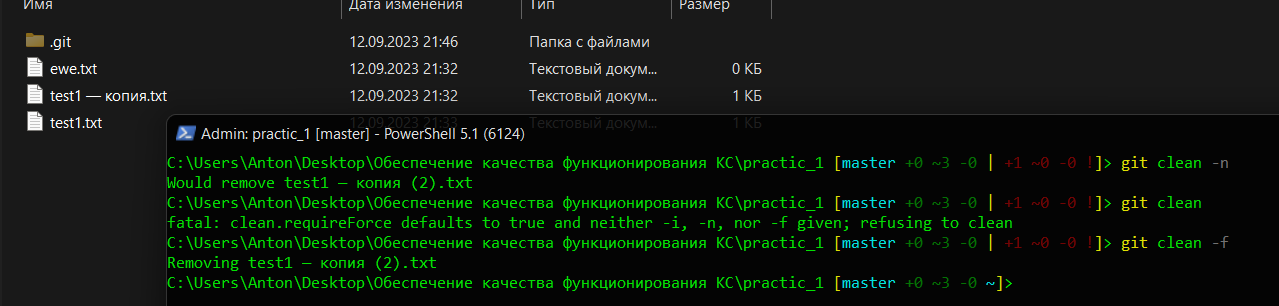


Рисунок 14. git clean.

Head – указатель на текущую ветку, которая указывает на последний коммит.

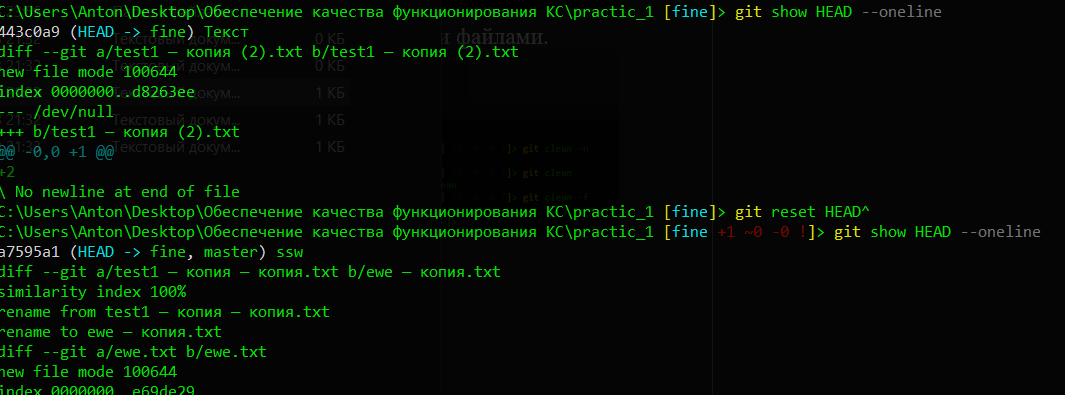


Рисунок 15. git show HEAD.

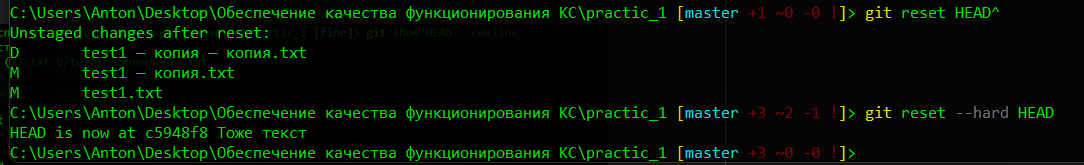


Рисунок 16. git reset HEAD.

Вывод: была выполнена поставленная задача. Была усвоена лекция. Были усвоены основные команды git и продемонстрированы с пояснением в практической.

# Практическая работа №2.

Цель работы: Работа Git-хостингами:

1. Конспект лекции

2. Описать создание аккаунта на любом Git-хостинге. Кратко описать функционал различных вкладок.

3. Работа с удаленным репозиторием. Основные команды для работы с удаленным репозиторием ( + обязательно сделать ограничение на определенную ветку и попробовать залить сразу несколько веток, описать результат)

Git-хостинги - это платформы, которые предоставляют инфраструктуру для хранения и управления репозиториями Git. Они

позволяют разработчикам работать с Git репозиториями удаленно,

управлять версиями, отслеживать изменения и сотрудничать с другими

разработчиками.

GitLab - это хостинг-платформа для хранения и управления кодом. Она предоставляет функциональность, аналогичную GitHub, но с открытым исходным кодом. GitLab может быть развернут на собственных серверах, что позволяет организациям иметь полный контроль над своими репозиториями. Он также предлагает широкий набор инструментов для управления проектами.

После нажатия кнопки регистрации на следующей странице нас просят написать почту, придумать пароль, имя пользователя и указать нужна ли нам рассылка и в конце пройти капчу и остаётся только нажать кнопку создания аккаунта.

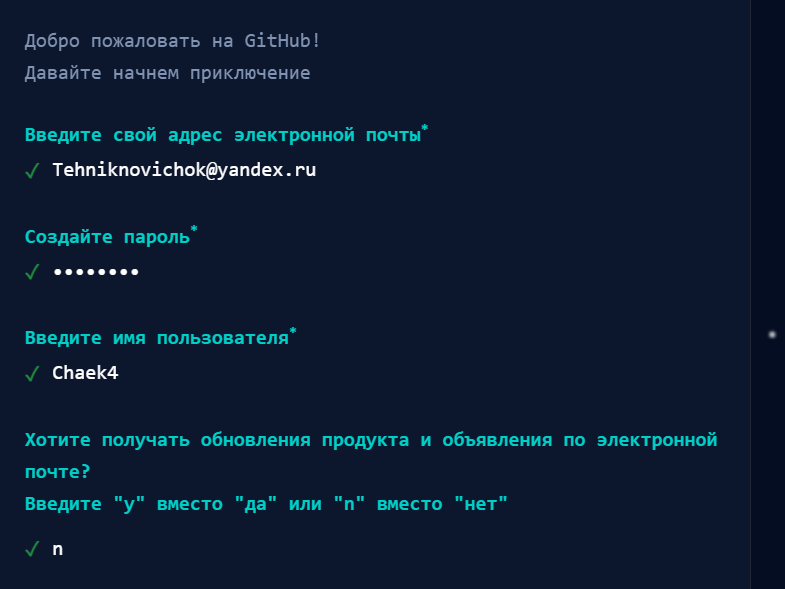


Рисунок 17. Заполнение данных.

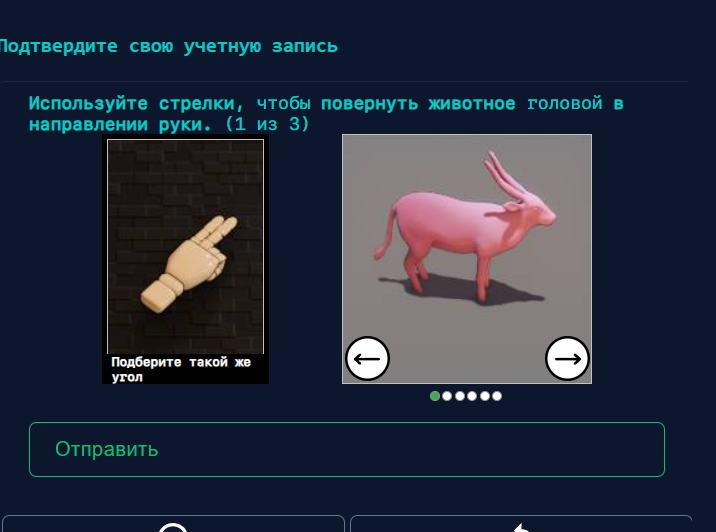


Рисунок 18. Капча.

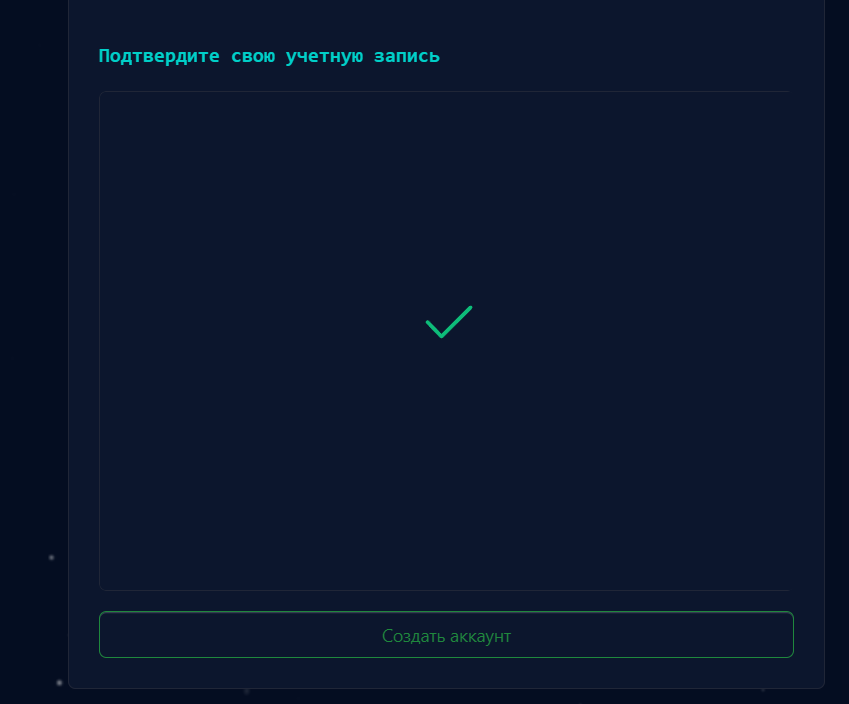


Рисунок 19. Пройденная капча.

В Github`е мы расмотрим 8 элементов бокового меню:

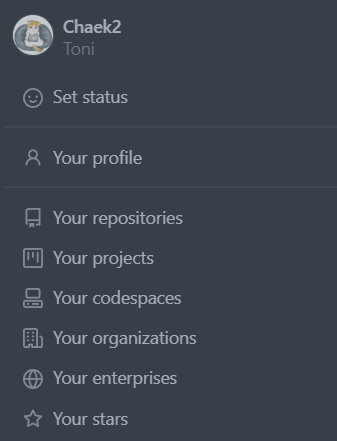


Рисунок 20. Элементы бокового меню.

Установка статуса: (текст, режим «не беспокоить», срок)

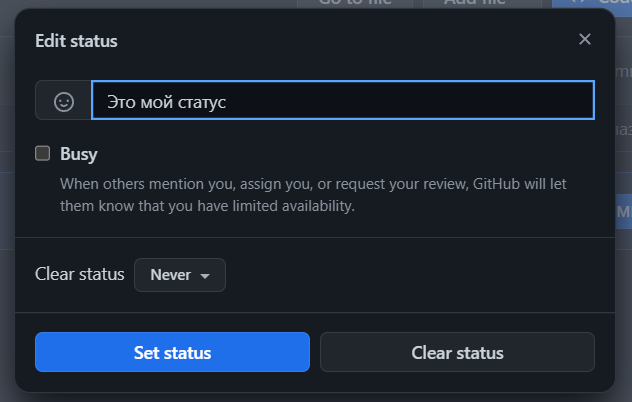


Рисунок 21. Окно статуса.

Мой профиль: (главная страница)

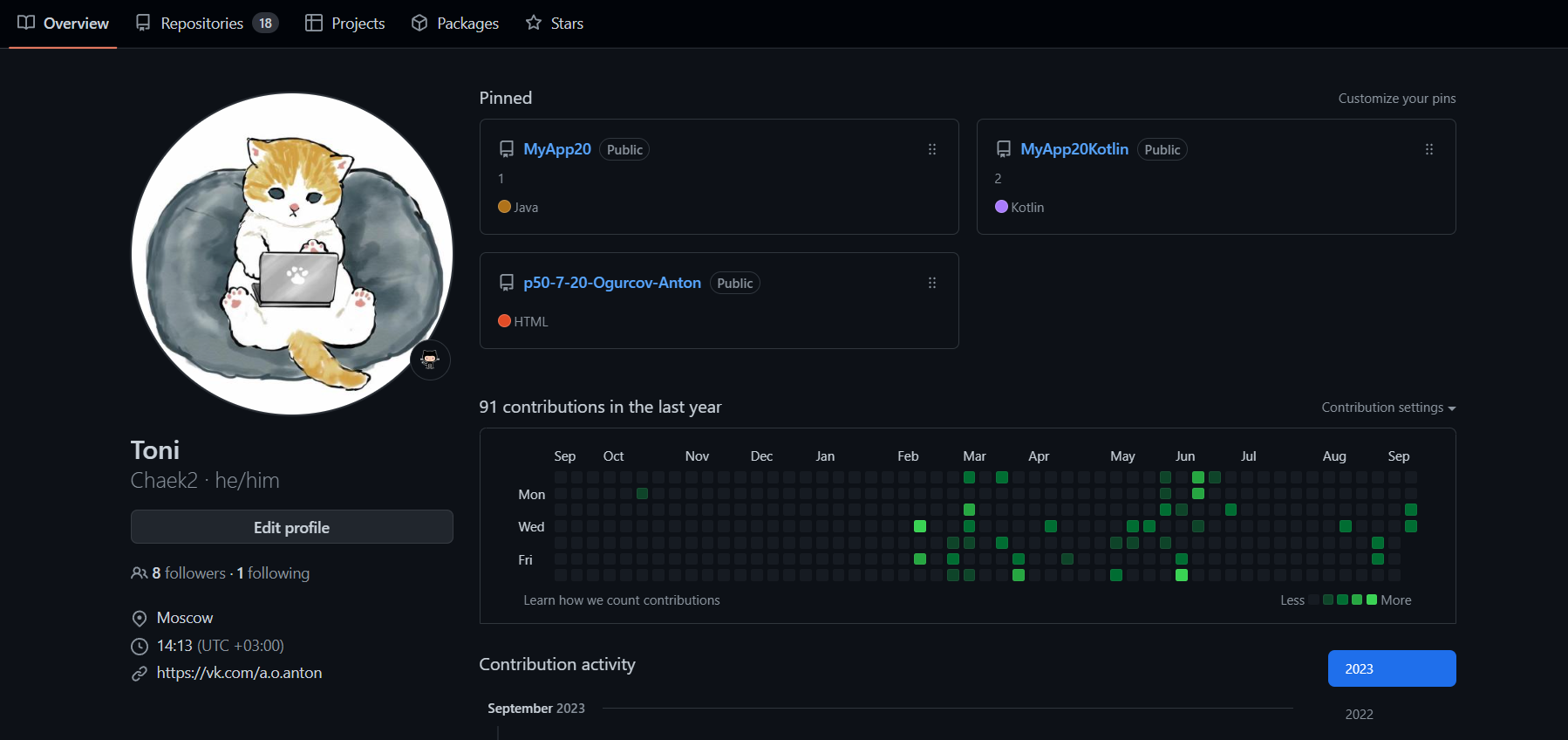


Рисунок 22. Главная страница.

Мои репозитории: (страница со всеми репозиториями)

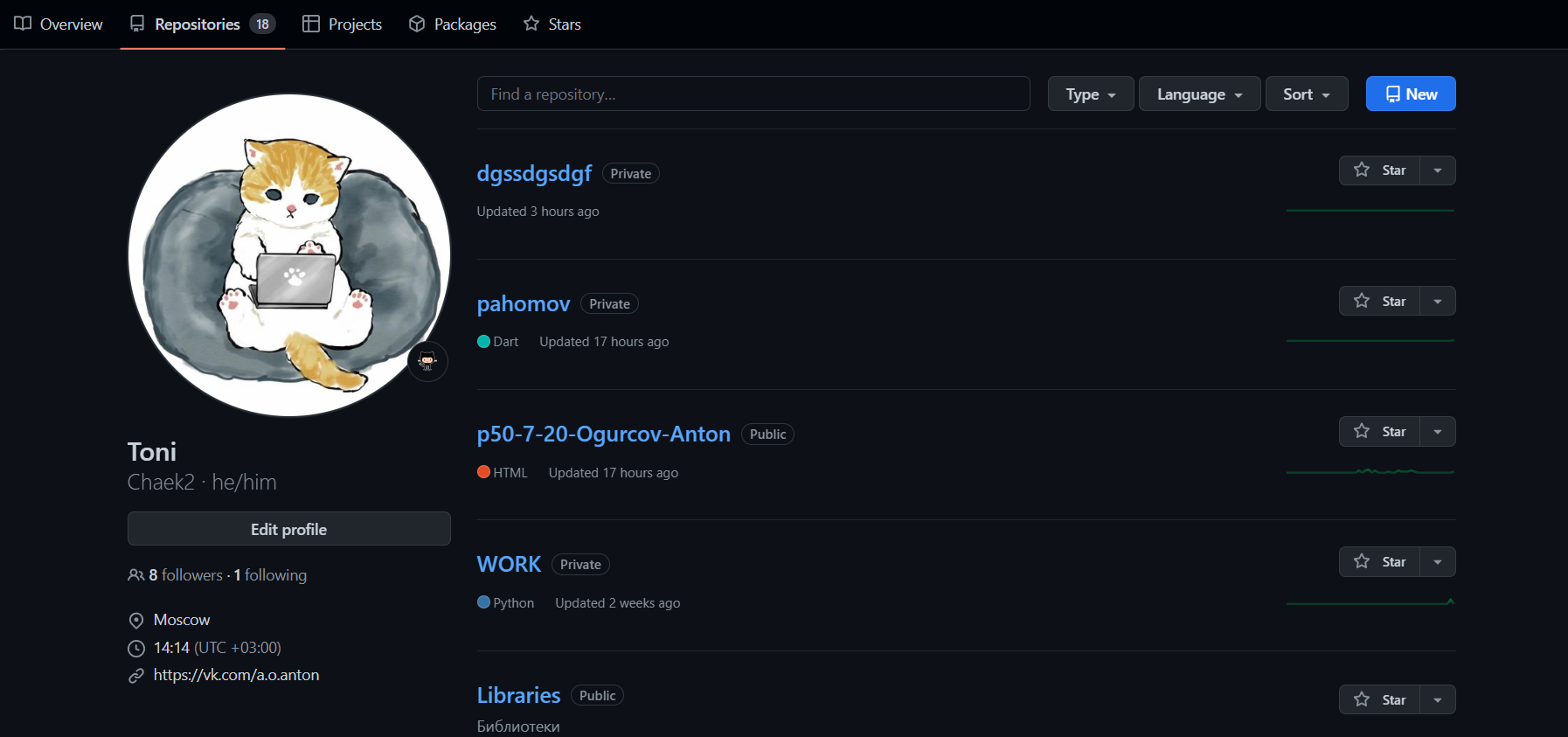


Рисунок 23. Репозитории.

Мои проекты: (страница со всеми проектами)

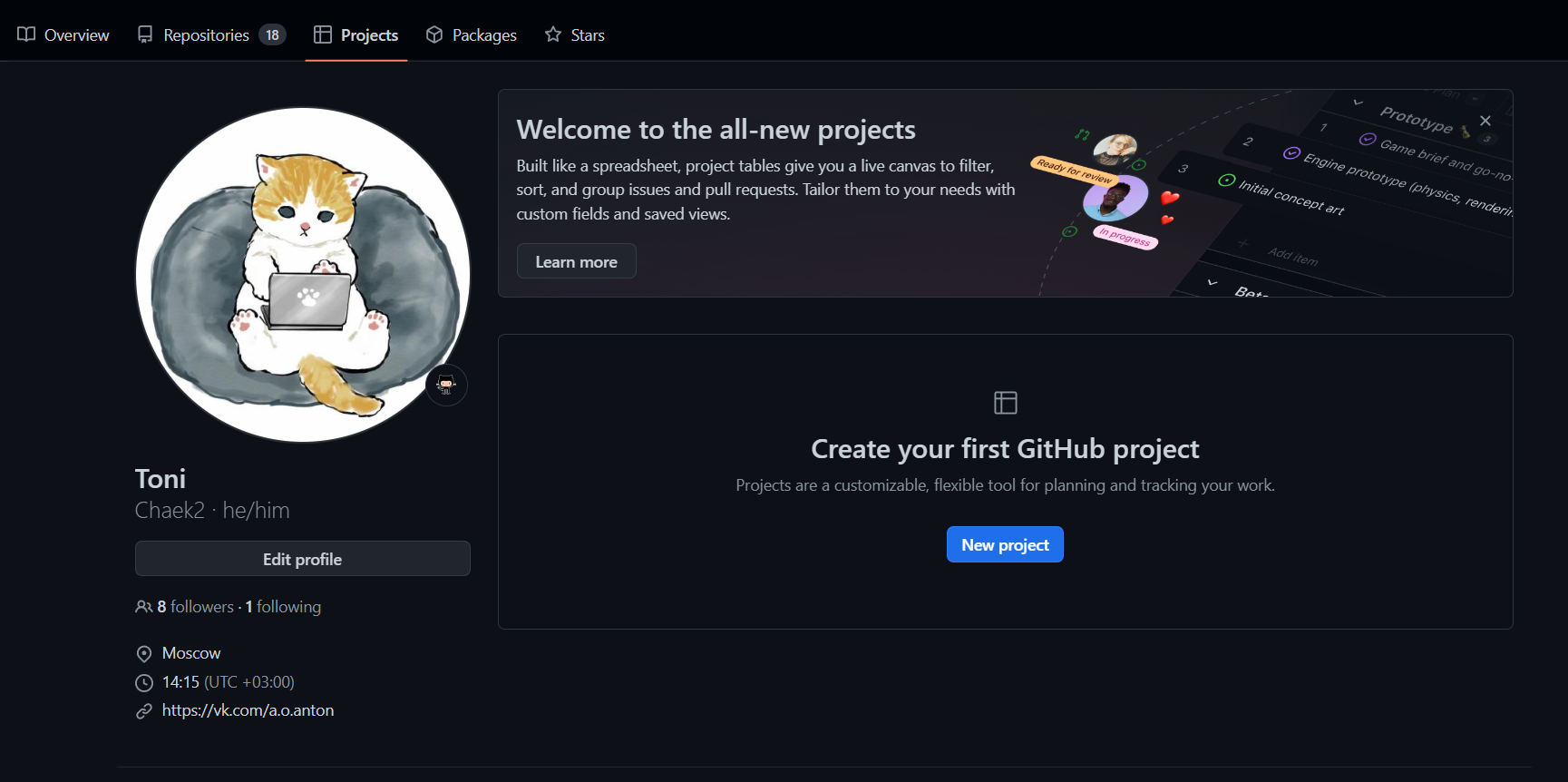


Рисунок 24. Проекты.

Мои среды разработки: (сервис основанный на Visual Studio Online, код можно редактировать прям в браузере)

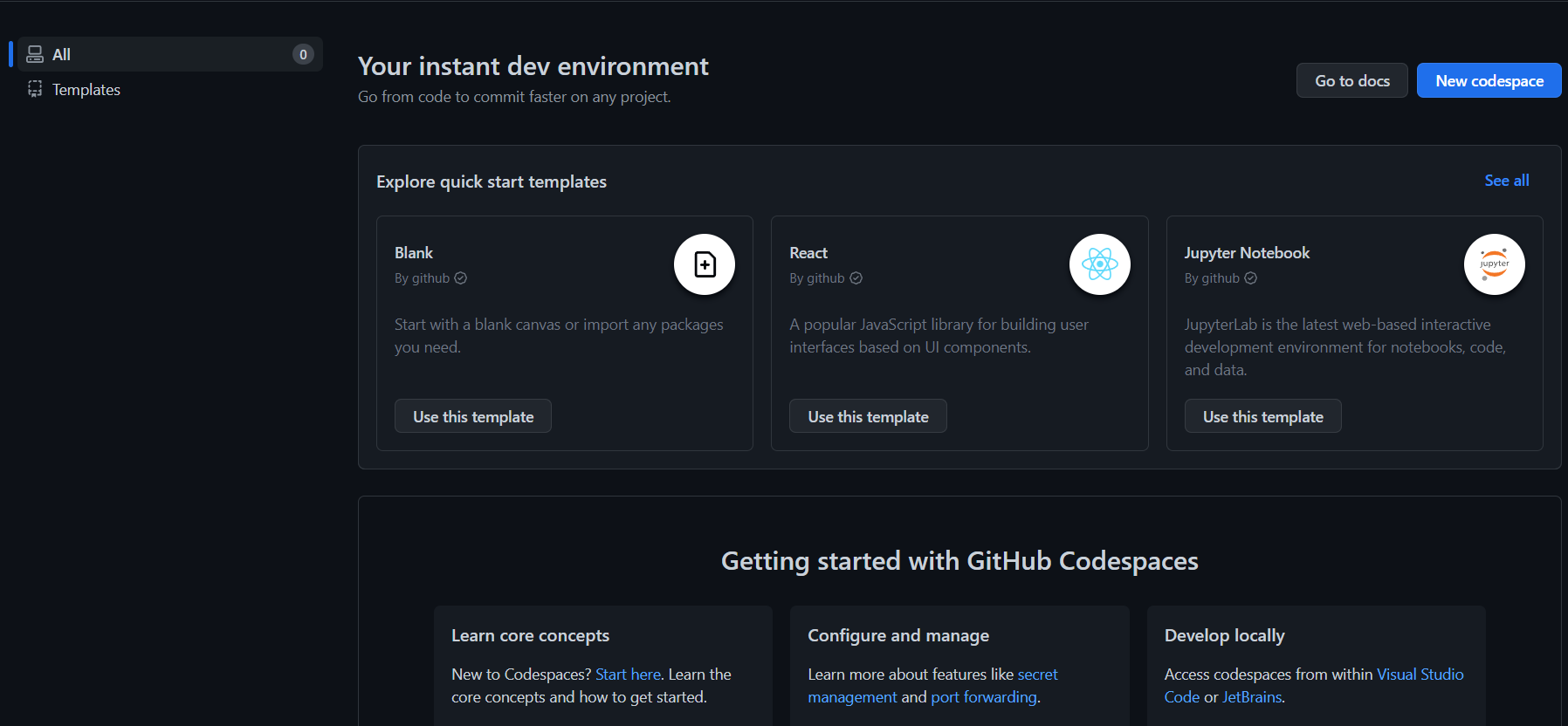


Рисунок 25. Онлайн разработка.

Мои организации:( страница выводящая информацию о связанных с вами организациях)

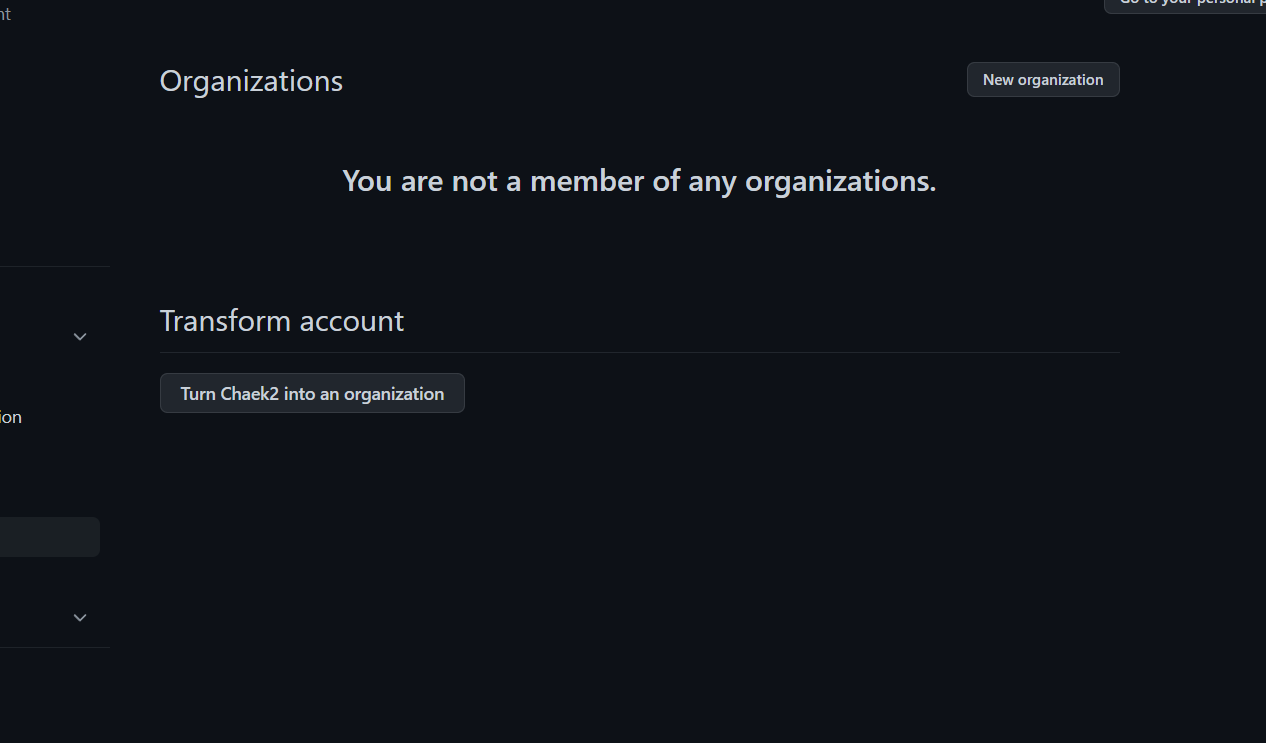


Рисунок 26. Организации.

Мои предприятия: (Предназначен для предприятий или команд, которые сотрудничают в GitHub.com)

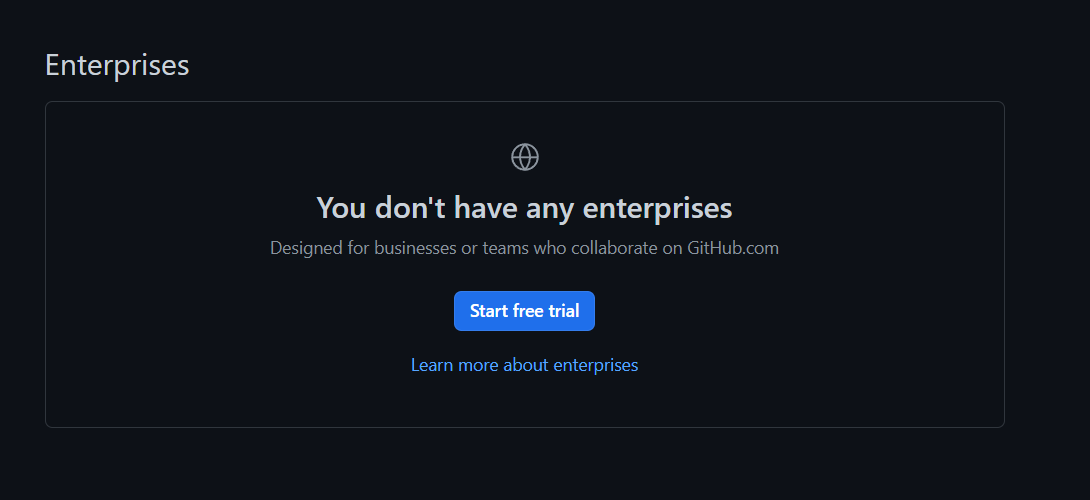


Рисунок 27. Предприятия.

Мои звёзды: (Список из помеченных репозиторий)

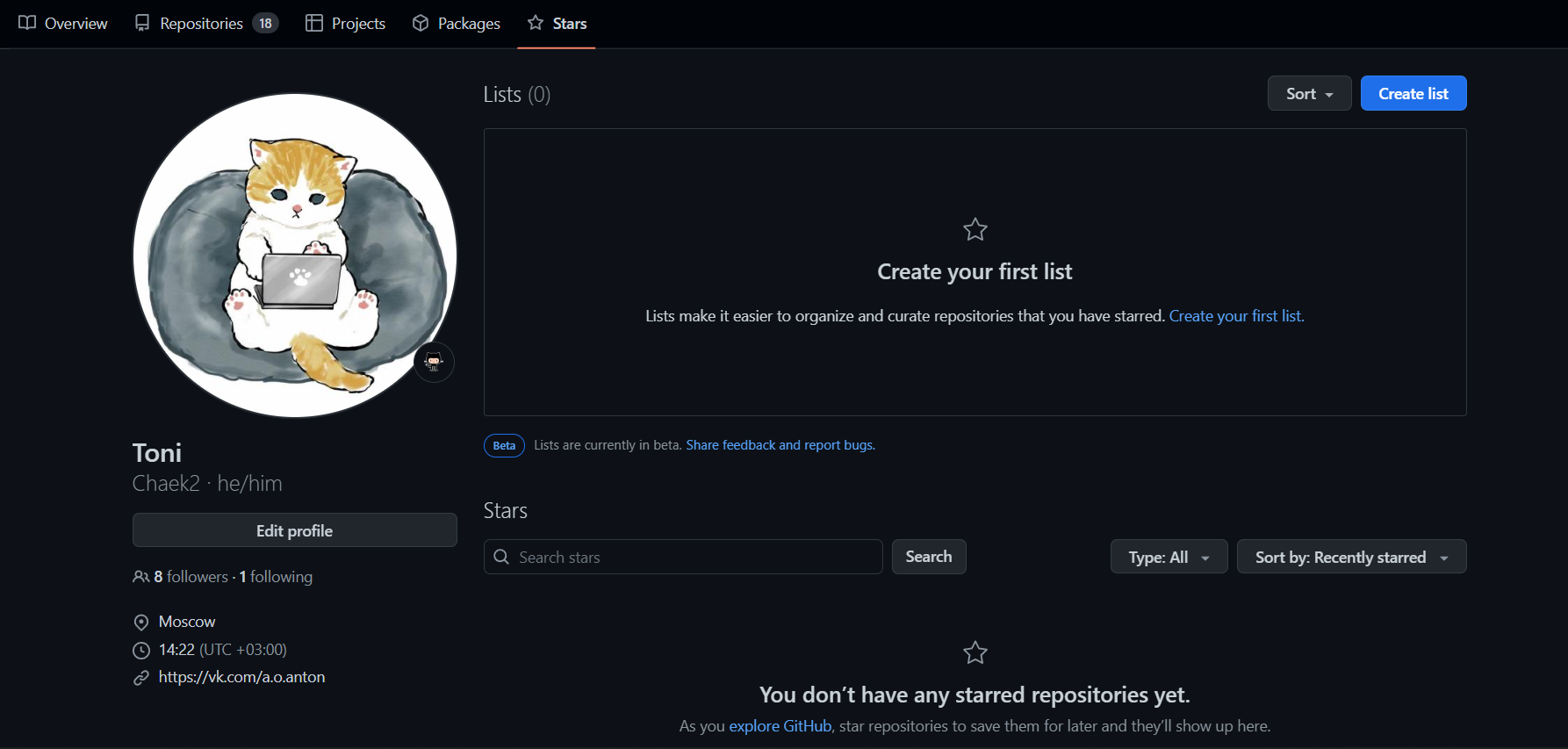


Рисунок 28. Избранные.

Из меню на главной странице ещё не рассмотрен Package: ( Пакеты ПО)

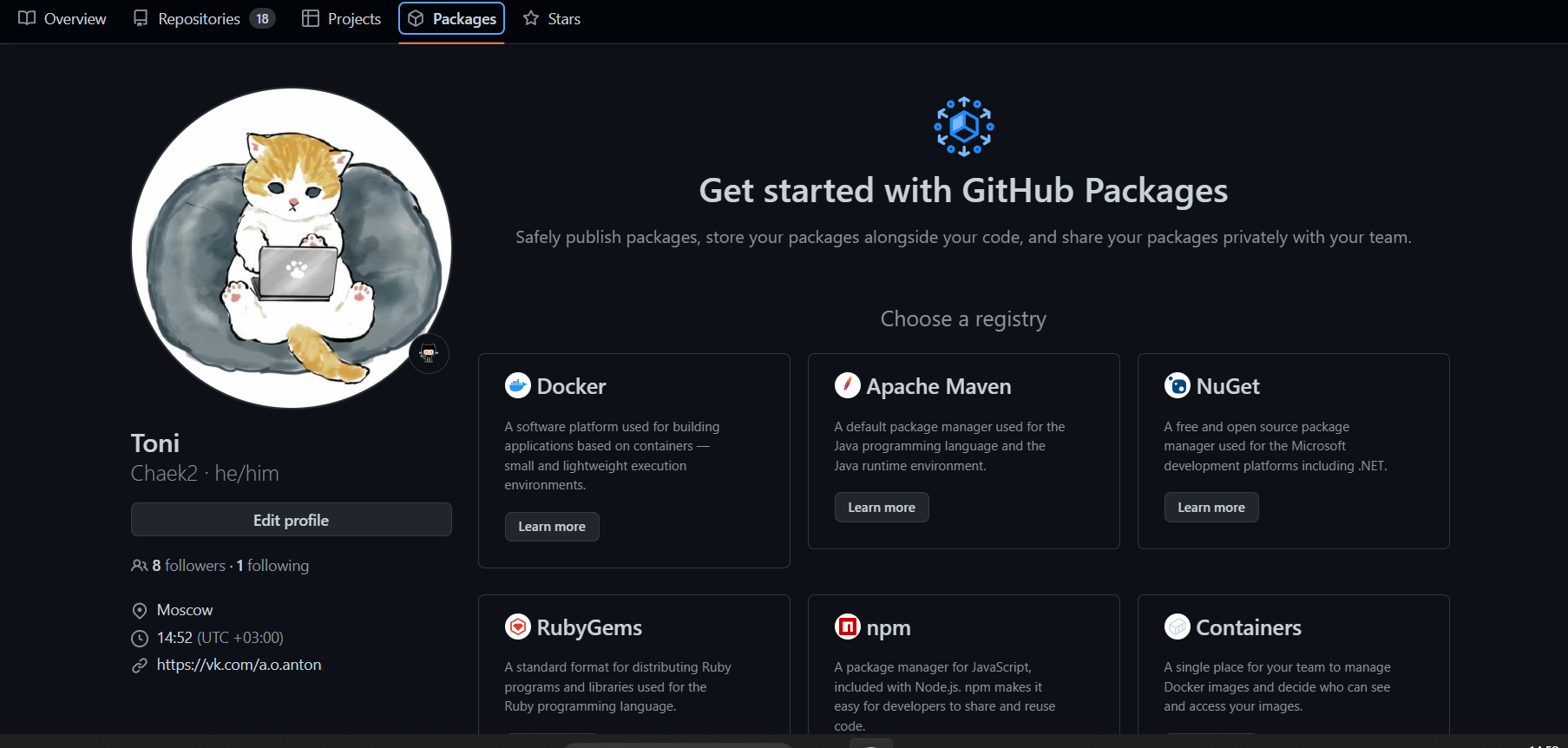


Рисунок 29. Пакеты.

Работа с удалённым репозиторием начинается с создания связи через git remote – команда, позволяющая подключаться к другим репозиториям.

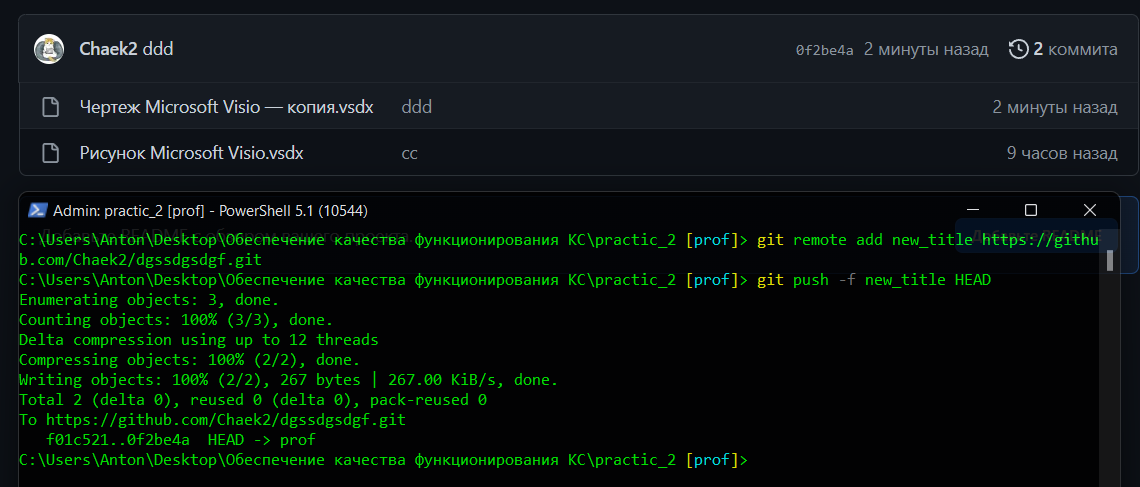


Рисунок 30. git remote.

Команда git clone позволяет скопировать репозиторий в текущую папку.

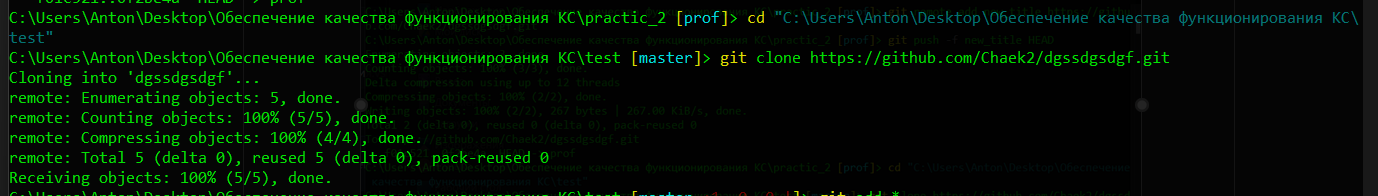


Рисунок 31. git clone.

Git push – команда, выгружает содержимое локального репозитория на удалённый. В отличии от fetch где коммиты импортируются в локальные ветки, то при помощи этой команде коммиты экспортируются в удаленные ветки



Рисунок 32. git push.

Git pull – вытягивание с сервера данных.

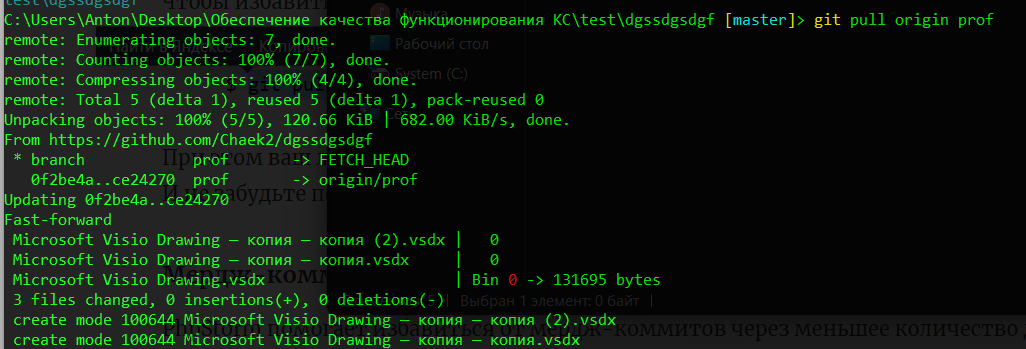


Рисунок 33. git pull.

Закрытие ветки на запись.

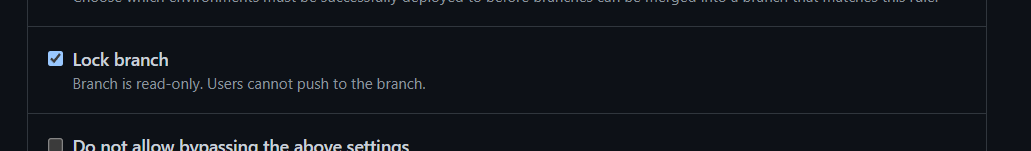


Рисунок 34. Github Lock.

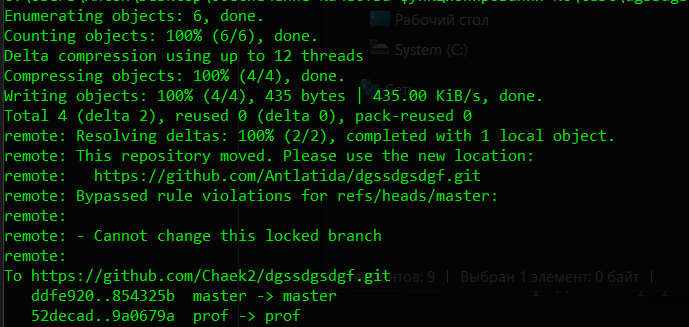


Рисунок 35. Push - Lock.

Вывод: Были выполнены поставленные цели. Был проконспектирован лекционный материал. Было показано создание собственного аккаунта на GitHub. Были продемонстрированы команды: git push, git pull, git remote.

# Практическая работа №3.

Цель работы: Начало работы с Docker:

1. Выполнить сборки и развертывание контейнеров из примера на паре, и описать своими словами то, что именно вы сделали и как.

2. Описать все вкладки в Docker (включая вкладки внутри собранного образа и контейнера)

3. Написать свои Doсkerfile и собрать контейнеры с программами из архива.

4. Залить готовую практическую ( все программы + Dockerfile к каждой из них) на Gitlab или Github и прикрепить к заданию ссылку.

Описать по всем пунктам со скриншотами, оформить в отчет. Все также, как и всегда - в ЕДИНОМ файле с ОГЛАВЛЕНИЕМ. Также не забываем о том, что в любом отчете есть Цель работы и вывод.

Лекция:

Docker - открытая платформа для разработки, доставки и запуска приложений. Он позволяет упаковать приложение и его зависимости в контейнер, который может быть запущен на любой совместимой с докер системой. Контейнеры докер предлагают независимо от операционной системы среду выполнения.

Преимущества использования в нашем программном обеспечении:

Изолированная среда. Докер обеспечивает изолированное окружение, в котором приложение может работать независимо от других приложений и зависимостей на хостовой системе. Это позволит избежать конфликтов и снижает вероятность неправильной работы приложения. Поэтому разработчики могут не задумываться в каком окружении будет работать их приложение, а инженеры по эксплуатации – единообразно запускать положение и меньше заботиться о системных зависимостях.

Управление зависимостями. Докер позволяет включать все зависимости приложения, включая операционную систему, библиотеки и другие компоненты, в контейнер. Это упрощает управление зависимостями и обеспечивает согласованность окружения между разработчиками и на разных этапах разработки и может снизить проблемы, связанные с развертыванием приложения на разных серверах.

Масштабируемость. Докер позволяет масштабировать приложение по мере необходимости, добавляя новые контейнеры или запуская их на кластере докер. Это обеспечивает гибкость и удобство в управлении нагрузкой.

Переносимость. Докер позволяет легко перемещать контейнер с приложением между различными средами разработки, тестирование и продакшн. Это упрощает процесс развертывания и масштабирования приложений.

Dockerfile – это текстовый файл, содержащий инструкции для автоматического создания докер-образа. Он описывает шаги, которые необходимо выполнить для создания докер-контейнера, включая установку и настройку приложений, копирование файлов и настройку сетей. Докер файл позволяет разработчикам создавать повторяемые и легко воспроизводимые среды разработки и выполнения приложений, где каждый шаг указан явно и может быть автоматически выполнен при создании контейнера докер.

Контейнеры – это изолированные и легковесные среды, в которых запускается приложения и их зависимости. Они представляют собой запускаемые экземпляры образов докер, содержащих все необходимые для работы приложения. Их можно создавать, запускать, останавливать и удалять. Также можно подключать к контейнеру хранилище, объединять контейнеры одной или несколькими сетями и общаться с контейнерами, используя докер апи или командную строку.

Образ – это шаблон контейнеров. В образе описывается, что должно быть установлено в контейнере (например код, среда выполнения, библиотеки, переменные окружения и другие зависимости) и какие действия нужно выполнить при старте контейнера.

Используя один образ можно создавать много контейнеров.

Кластер докер – это совокупность компьютеров или серверов, которые объединены вместе для работы с контейнерами докер. Кластер докер использует контейнеры для запуска и управления приложением на нескольких компьютерах одновременно. Кластер обеспечивает масштабированность и надежность, по скольку если один компьютер выходит из строя, другие компьютеры в кластере могут продолжать работать без проблем. Кластер также позволяет распределить нагрузку между компьютерами, чтобы обеспечить эффективное использование ресурсов.

Докер компос – инструмент, который позволяет запускать и управлять несколькими контейнерами докер одновременно. Он позволяет описывать структуру и конфигурацию вашего приложения в файле YAML, что делает его очень простым для понимания и использования. Докер компос позволяет запускать и остановить все ваши контейнеры одной командой, а также связывать их между собой, чтобы они могли взаимодействовать друг с другом.

Докер демон – основной компонент в системе докер, который управляет выполнениями контролем контейнеров. Он работает в фоновом режиме и управляет ресурсами, созданием и удалением контейнеров, а также назначением им ресурсов.

Докер клиент – инструмент командной строки, который позволяет управлять контейнерами докер (через него пользователь взаимодействует с демон) позволяет выполнять различные операции с контейнерами, такие как создание, запуск, остановка, удаление и мониторинг, используя простые команды в командной строке.

Команды для работы с докерами.

Docker info – получение информации и докер.



Рисунок 36. Синтаксис: docker info

Docker version – вывод информации о версии докер.



Рисунок 37. Синтаксис: docker version

Docker ps – просмотр активных контейнеров.



Рисунок 38. Синтаксис: docker ps

FROM – указывает базовый образ, на основе которого будет создаваться новый образ.



Рисунок 39. Синтаксис: From

LABEL – добавляет метаданные к образу, такие как описание версия, автор и др.



Рисунок 40. Синтаксис: label

RUN – выполняет команду внутри образа и создает слой с результатами выполнения.



Рисунок 41.Синтаксис: run

COPY – копирует файлы или дирректории из хост-системы в образ.



Рисунок 42. Синтаксис: copy

ADD – подобно COPY, но позволяет также скачивать файлы из интернета и распаковывать архивы.



Рисунок 43. Синтаксис: add



Рисунок 44. Синтаксис: add . Часть 2

WORKDIR – устанавливает рабочий каталог для следующих инструкций.



Рисунок 45. Синтаксис: workdir

ENV – устанавливает переменные среды в образе.



Рисунок 46. Синтаксис: env

EXPOSE – объявляет порты, на которых приложение внутри контейнера слушает входящие соединения.



Рисунок 47. Синтаксис: expose

CMD – Устанавливает команду по умолчанию, которая будет выполняться при запуске контейнера. Может быть переопределено при запуске контейнера.



Рисунок 48. Синтаксис: cmd

ENTRYPOINT – устанавливает точку входа для контейнера, определяя исполняемую команду. Может быть переопределено при запуске контейнера.



Рисунок 49. Синтаксис: entrypoint

Docker build – создание образа из Dockerfile.



Рисунок 50. Синтаксис: docker build

Docker run – запуск контейнера из образа.



Рисунок 51. Синтаксис: docker run

Docker stop – остановка контейнера.



Рисунок 52. Синтаксис: docker stop

Docker restart – перезапуск контейнера.



Рисунок 53. Синтаксис: docker restart

-t или –tag – установить имя и опциональный тег для образа.

--build-arg – установить сборочные аргументы, доступные в Dockerfile.

-f или –file – указать имя Dockerfile. Если оно отличается от стандартного имени (Dockerfile).

--no-cache – запустить сборку с отключением кеширования промежуточных слоев.

--rm – удалить временные контейнеры после сборки образа.

-d или –detach – запустить контейнер в фоновом режиме.

-p или –publish – перенаправляет порты контейнера на порты хоста.

-v или –volume – создать точку монтирования для доступа к файлам или каталогам на хосте.

--name – указать имя контейнера.

--env или -e –установить переменные окружения в контейнере.

--network – подключить контейнер к определенной сети.

--restart – определяет, как контейнер должен быть перезапущен в случае сбоя.

--rm – удаляет контейнер после его завершения.

-it – вход в контейнер с интерактивными терминалом.

1. Выполнить сборки и развертывание контейнеров из примера на паре, и описать своими словами то, что именно вы сделали и как.

Создаём файл с расширением py и пишем какой-нибудь рабочий код.



Рисунок 54. Пример файла на паре.

Создаём Dockerfile в котором указывает команды, который docker понимает.

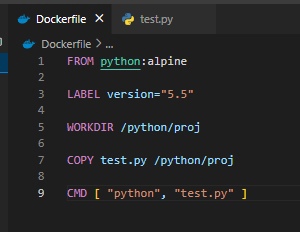


Рисунок 55. Пишем свой Dockerfile

Создаём контейнер с уникальным именем.

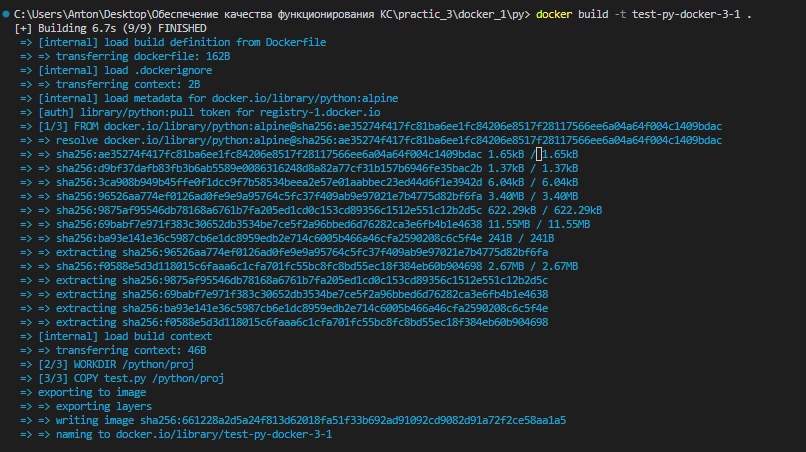


Рисунок 56. Сборка docker контейнера.

Запускаем контейнер.



Рисунок 57. Пример контейнера. Запуск.

Если зайти в программу, то увидим наш контейнер.



Рисунок 58. Фото из программы.

1. Описать все вкладки в Docker (включая вкладки внутри собранного образа и контейнера)

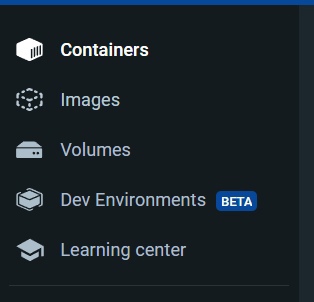


Рисунок 59. Вкладки главного меню.

Containers – Представление контейнеров предоставляет представление среды выполнения всех ваших контейнеров и приложений.

Images – В представлении образ отображается список ваших изображений Docker и позволяет запускать образ в качестве контейнера, извлекать последнюю версию образа из Docker Hub и просматривать изображения.

Volumes – Представление томов отображает список томов и позволяет вам легко создавать и удалять тома, а также видеть, какие из них используются.

Dev Environments – среды разработки

Docker Scout – инструмент для управления и мониторинга контейнерами

Learning center – Просмотр учебного центра поможет вам начать работу с быстрыми пошаговыми инструкциями в приложении и другими ресурсами для изучения Docker.

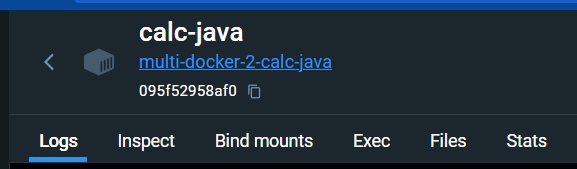


Рисунок 60. Меню контейнера.

Logs – Список логов

Inspect – Низкоуровневые сведения о контейнере

Bind mounts – привязка креплений

Exec – Консоль

Files – Список файлов

Stats – Статистика

1. Написать свои Doсkerfile и собрать контейнеры с программами из архива.



Рисунок 61. Dockerfile для calc.java

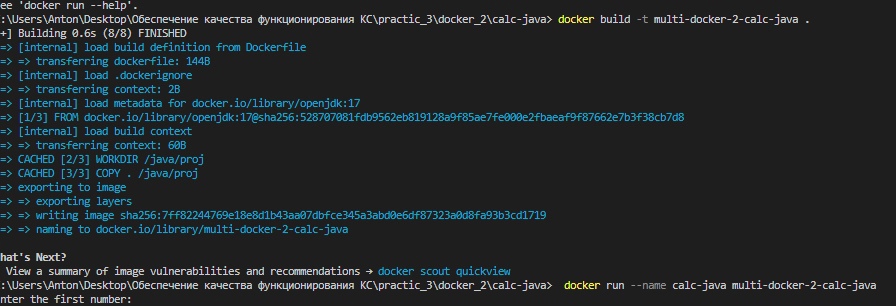


Рисунок 62. Создание и запуск контеёнера calc.java.



Рисунок 63. Dockerfile для game.java

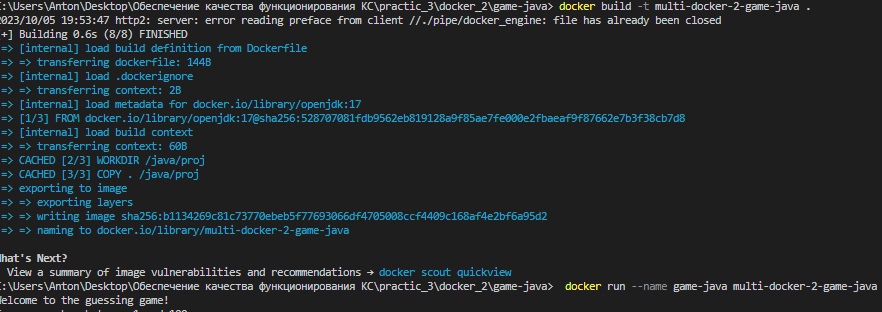


Рисунок 64. Создание и запуск контеёнера game.java

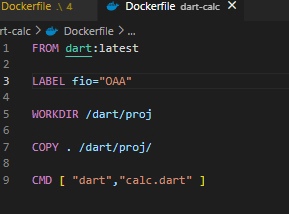


Рисунок 65. Dockerfile для calc.dart

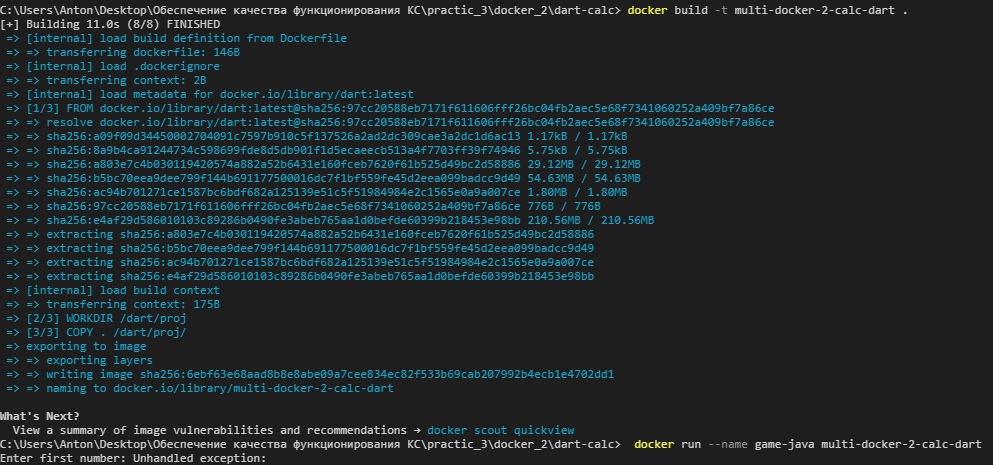


Рисунок 66. Создание и запуск контеёнера calc.dart



Рисунок 67. Dockerfile для calc.py

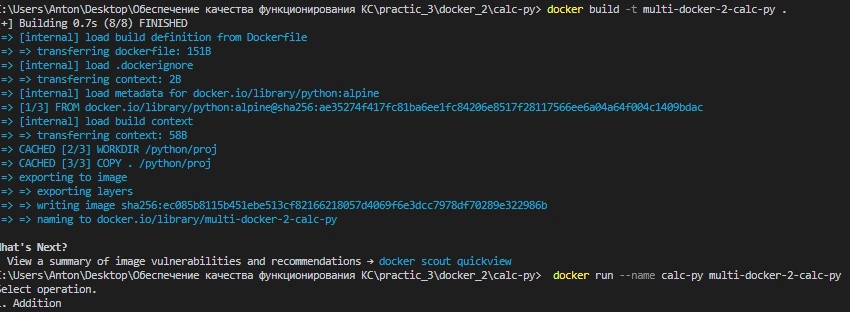


Рисунок 68. Создание и запуск контеёнера calc.py

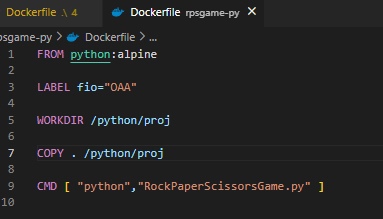


Рисунок 69. Dockerfile для game.py

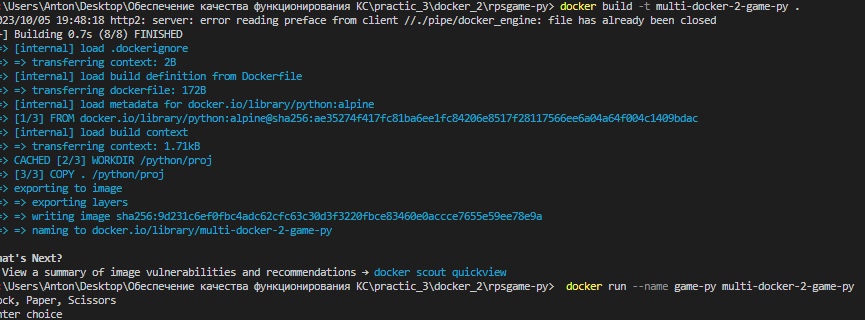


Рисунок 70. Создание и запуск контеёнера game.py

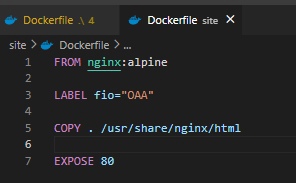


Рисунок 71. Dockerfile для site's

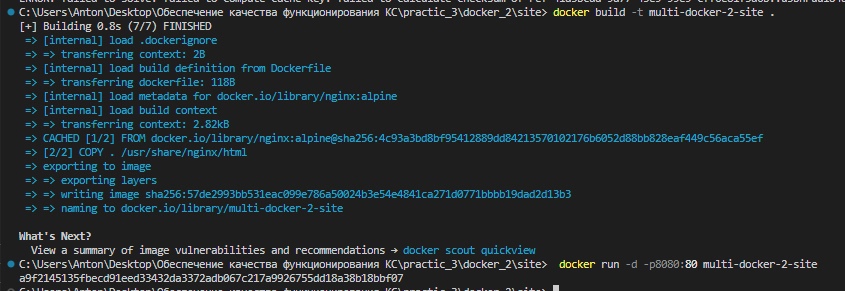


Рисунок 72. Создание и запуск контеёнера site's

1. Залить готовую практическую ( все программы + Dockerfile к каждой из них) на Gitlab или Github и прикрепить к заданию ссылку.

Github: <https://github.com/Chaek2/bullying/tree/master/practic_3>

Вывод: Были выполнены поставленные цели. Был проконспектирован лекционный материал. Материал загружен на удалённый репозиторий.

# Практическая работа №4.

Цель работы: Развертывание приложений в Docker

На 4:

1. Развернуть приложение time-db из примера в Docker. Описать каждый пункт -почему именно эти команды и компоненты вы создаете.

2. Залить готовую практическую на Gitlab или Github и прикрепить к заданию ссылку.

На 5:

1. Развернуть приложение time-db из примера в Docker. Описать каждый пункт -почему именно эти команды и компоненты вы создаете.

2. Залить готовую практическую на Gitlab или Github и прикрепить к заданию ссылку.

3. Развернуть любое свое приложение в Docker которое имеет как минимум 3 компонента (например API+База данных+Adminer. Но если ваше приложение будет без фронтенда , а например только с API, то вы должны будете подключиться к этой АПИ из своего приложения, не развернутого в Docker (например с мобилки)).

Лекция:

Docker Compose - это инструмент, который позволяет запускать и управлять несколькими контейнерами Docker одновременно. Он позволяет описывать структуру и конфигурацию вашего приложения в файле YAML, что делает его очень простым для понимания и использования. Docker Compose позволяет запускать и остановить все ваши контейнеры одной командой, а также связывать их между собой, чтобы они могли взаимодействовать друг с другом.

Декларативное программирование — это парадигма, при которой описывается желаемый результат, без составления детального алгоритма его получения.

YAML (YAML Ain't Markup Language) - это удобный формат для хранения и передачи данных в текстовой форме. Он используется для конфигурационных файлов, данных, и другой информации, которую читает и записывает компьютерное программное обеспечение. YAML напоминает человеко-читаемый текст и позволяет организовать данные в виде списков, словарей и других структур для удобства чтения и редактирования как людьми, так и компьютерами. Важно соблюдение табуляции (!!!)

Преимущества Docker compose: Все необходимые контейнеры запускаются и выключаются одной командой, Автоматическое создание необходимых образов на основании Dockerfile каждого приложения, Автоматическое создание изолированной сети для взаимодействия контейнеров, C помощью DNS возможно взаимодействие между контейнерами, используя имя.

version - Объявляет версию docker-compose синтаксиса.

services - Определяются все контейнеры, которые вы хотите

запустить с помощью docker-compose. Каждый сервис имеет

имя и определенные настройки.

Каждый сервис можно отдельно настраивать. image Можно использовать существующие Docker-образы из Docker Hub или указать путь к своему Docker-образу.

Можно настроить различные параметры для каждого сервиса, такие как переменные окружения, порты, названия контейнера и тд. Переменные окружения в Docker Compose позволяют определить и передать конфигурационные параметры в контейнеры.

depends\_on Можно указать зависимости между сервисами, чтобы гарантировать, что один сервис запускается только после запуска другого.

volumes Можно указать тома, чтобы сохранить данные между

запусками контейнеров.

Все, что находится в volumes сохраняется на диске после удаления контейнера. Остальные данные хранятся в памяти.

docker-compose up Запуск приложения в соответствии с конфигурацией из файла docker-compose.yml. Если файл называется иначе, добавляется ключ docker-compose -f <имя файла>

docker-compose down Остановка и удаление всех контейнеров,

сетей и томов, созданных docker-compose up.

docker-compose ps Отображение статуса контейнеров, запущенных с помощью Docker Compose.

docker-compose logs Просмотр логов контейнеров.

docker-compose build Построение образов, определенных в docker-compose.yml.

docker-compose exec <service-name> <command> Выполнение команды внутри контейнера.

Скачиваем архив с программами.

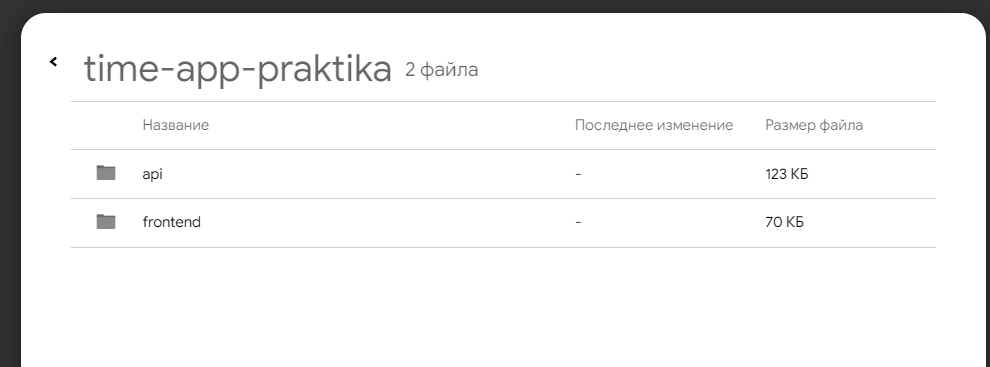


Рисунок 73. Архив в классруме.

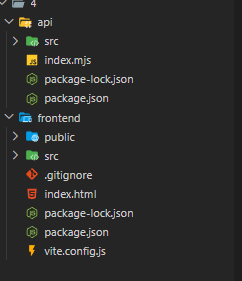


Рисунок 74. Архив в VS Code.

Пишем Dockefile для api. Указываем зависимость node для node.js.

Указываем папку, копируем пакеты и npm, устанавливаем порт и запуск через cmd, через рабработчика “dev”.

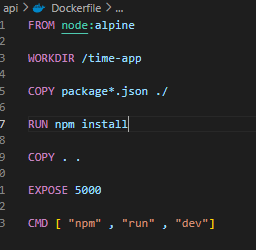


Рисунок 75. Dockerfile api.

Создаём такой же файл только меняем порт для сайта.

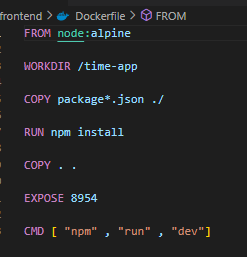


Рисунок 76. Dockerfile frontend.

В docker-compose указываем версию, сервисы: mysql, adminer, наш api, наш frontend; и указываем какие данные используем, т.е. данные бд храняшие по пути который мы прописали в сервисе mysql. В каждом сервисе указываем перезагрузку, порт, для api и frontend build, для api указываем зависимость от mysql.

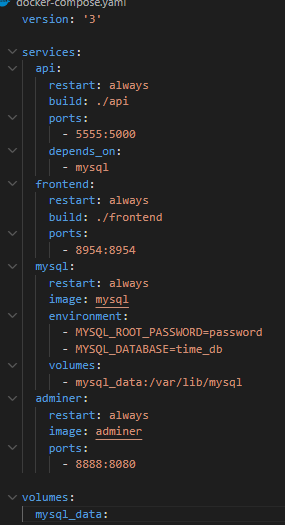


Рисунок 77.

Рисунок 78. Docker-compose.

Результат:

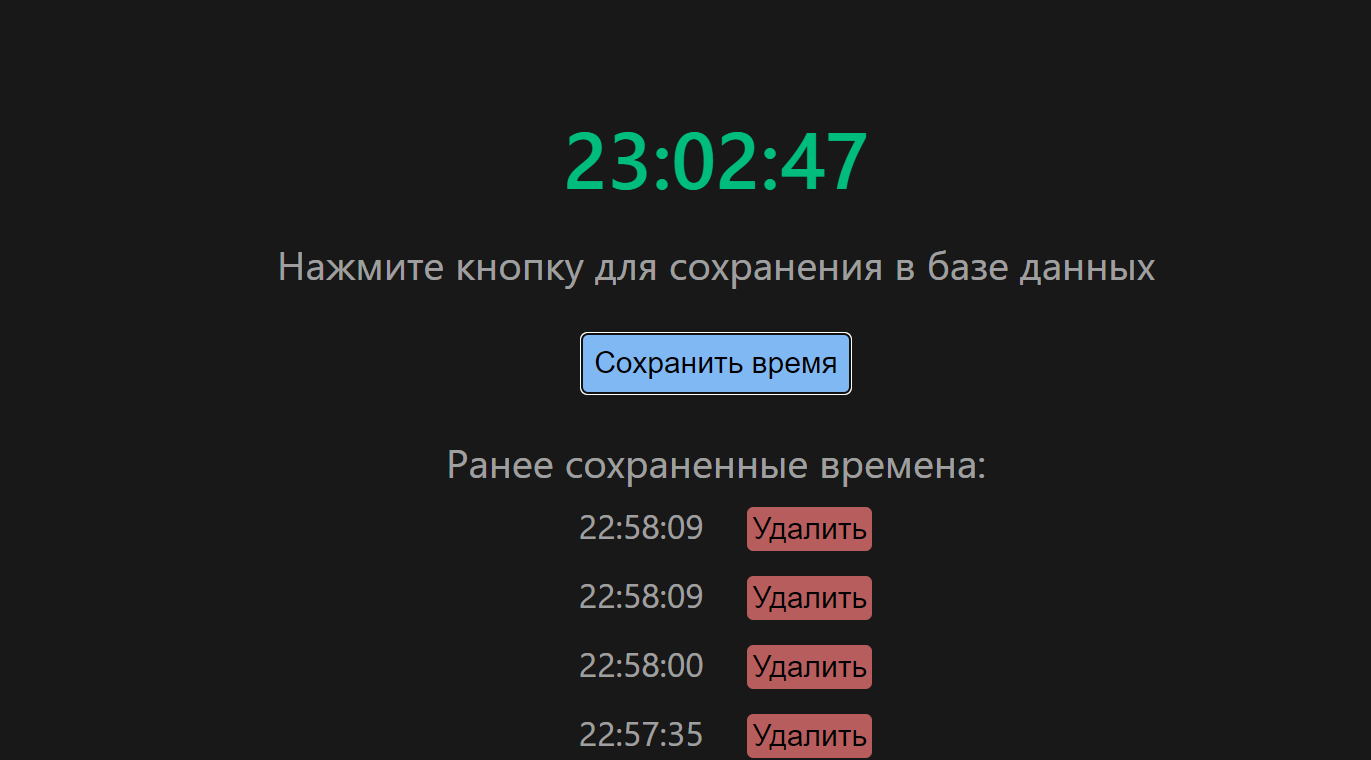


Рисунок 79. Сайт.

Результат adminer’а:



Рисунок 80. Строки в бд.

Создание своего контейнера.

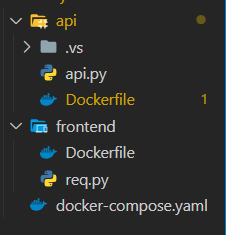


Рисунок 81. Иерархия проекта.

Были выбран язык python с библиотекой Flask для Api и создания сайта.

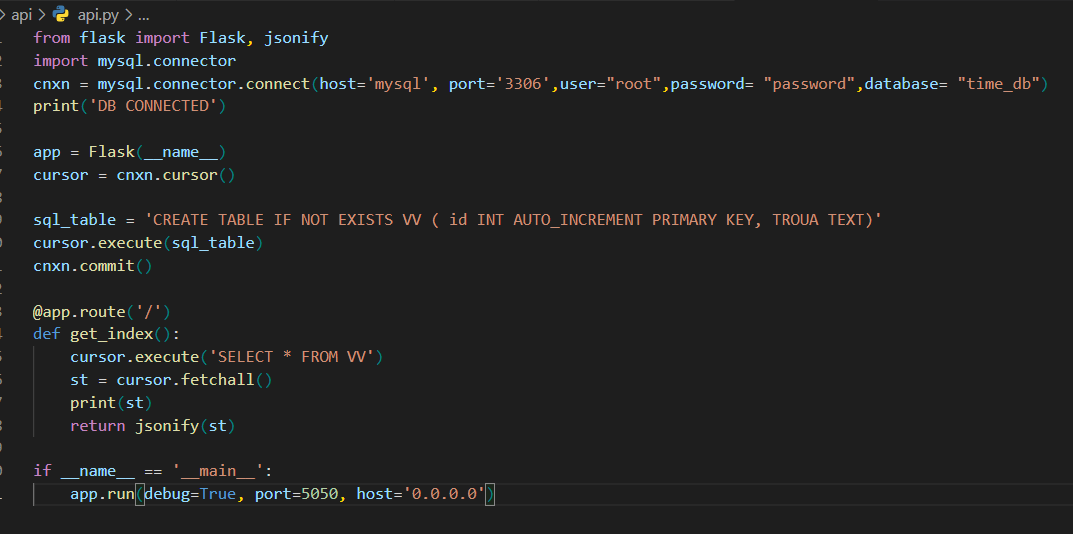


Рисунок 82. Код api.

Через команду run устанавливаем библиотеки.

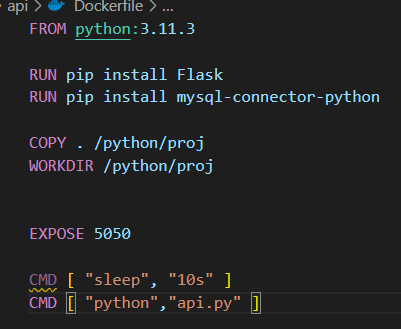


Рисунок 83. Dockerfile api.

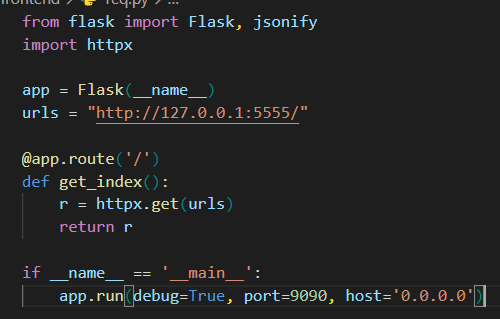


Рисунок 84. Код сайта.



Рисунок 85. Dockerfile сайта.

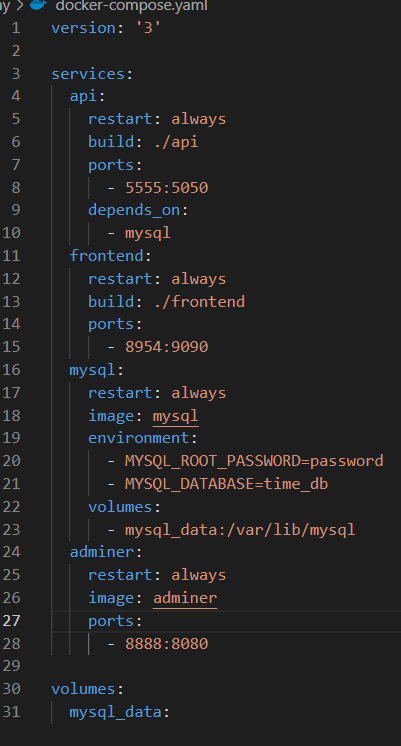


Рисунок 86. Docker-compose.

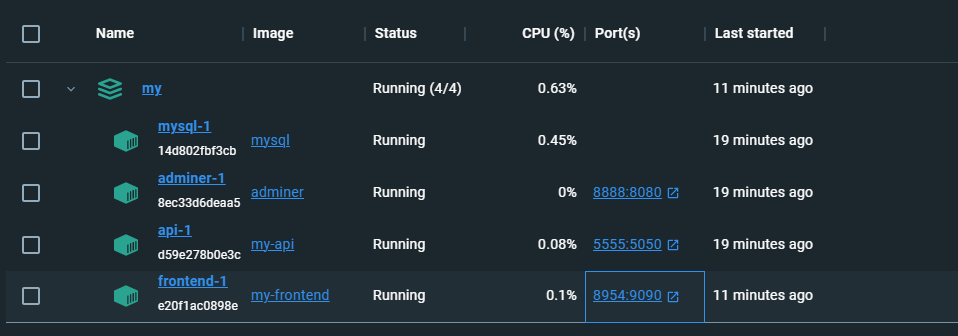


Рисунок 87. Контейнер.

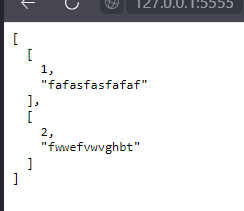


Рисунок 88. Ввод данных через api.



Рисунок 89. Записи таблицы.

Вывод: Были выполнены поставленные цели. Был проконспектирован лекционный материал. Материал загружен на удалённый репозиторий.

# Практическая работа №5.

Цель работы: Развертывание оконных приложений в Docker

1. Развернуть приложения xeyes и calc.jar из примера в Docker. Описать каждый пункт -почему именно эти команды и компоненты вы создаете.

2. Залить готовую практическую на Gitlab или Github и прикрепить к заданию ссылку.

Скачиваем файл .jar и создаём dockerfile.

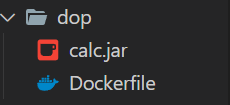


Рисунок 90. Иерархия.

Скачиваем зависимости и прописываем «адрес» для запуска окна.

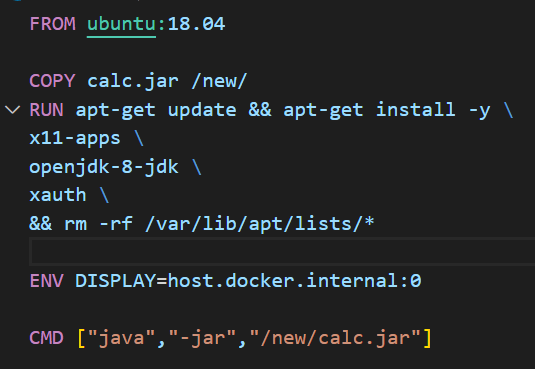


Рисунок 91. Dockerfile.

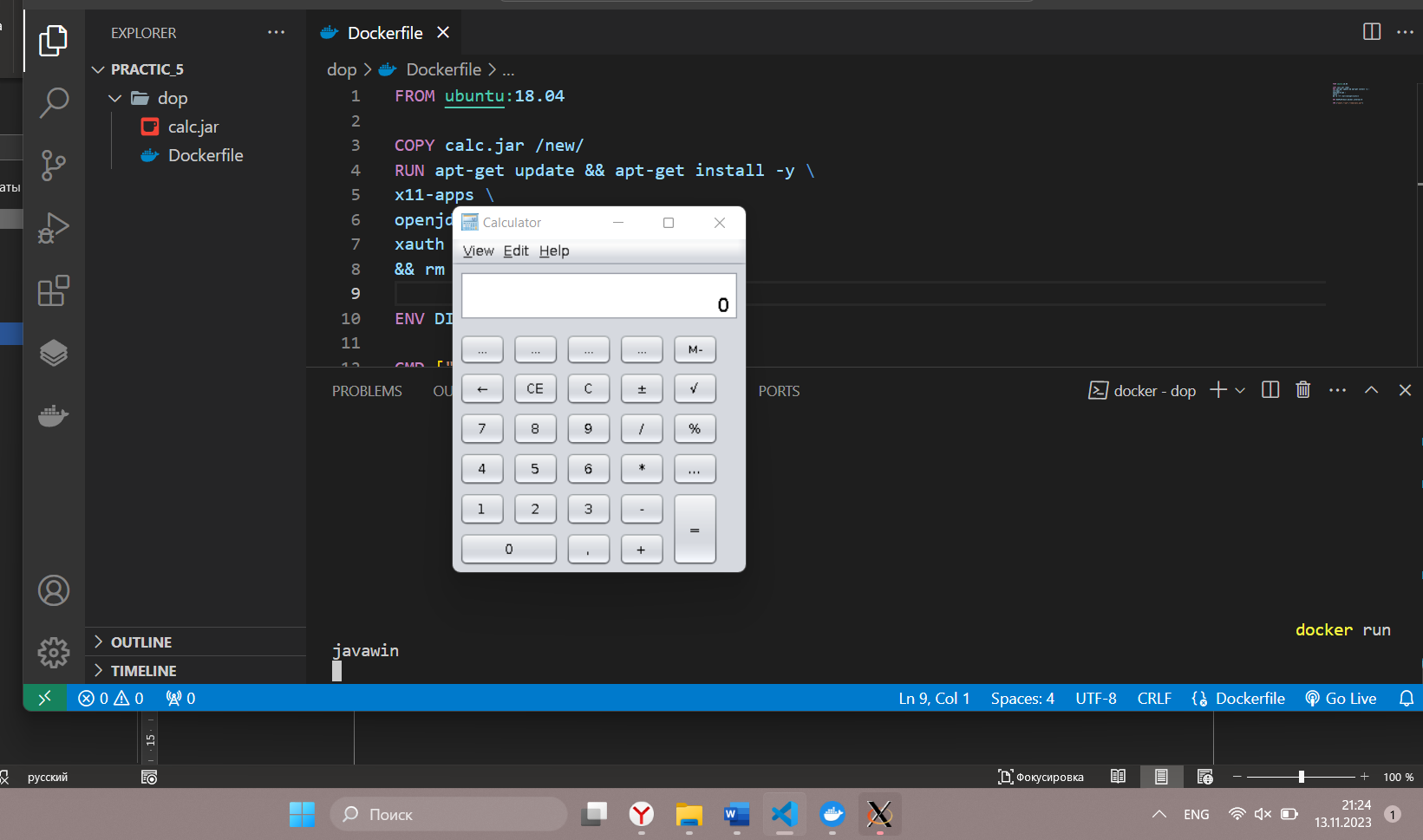


Рисунок 92. Запущенный контейнер.

Вывод: Была выполнена поставленная задача. Были получены новые знания и усвоены.

# Практическая работа №6.

Цель работы: Развертывание приложения с обновлением данных.

На 4:

1. Развернуть приложение на ASP.Net Core, которое меняет изображения через Bind Mounts и через Volume. Описать работу приложения, как работают данные компоненты, а также разницу между ними.

На 5:

2. Развернуть любое приложение НЕ на ASP.Net Core, которое будет обновлять какую-либо информацию через Bind Mounts и через Volume. Описать работу приложения, как работают данные компоненты, а также разницу между ними. Также можно сделать и на ASP.Net Core, но тогда обновляться должна не картинка, а что-нибудь другое (например css или текст из файла).

Создание проекта ASP.NET (MVC) в Visual Studio.

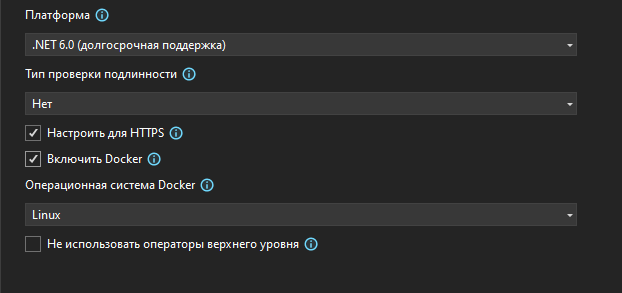


Рисунок 93. Создание проекта asp.net.

Добавляем папку с картинками.

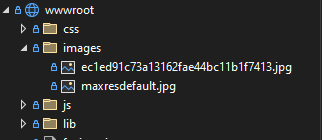


Рисунок 94. Файлы в проекте.

Запускаем с настройкой тома (volume).



Рисунок 95. Запуск с созданием тома.

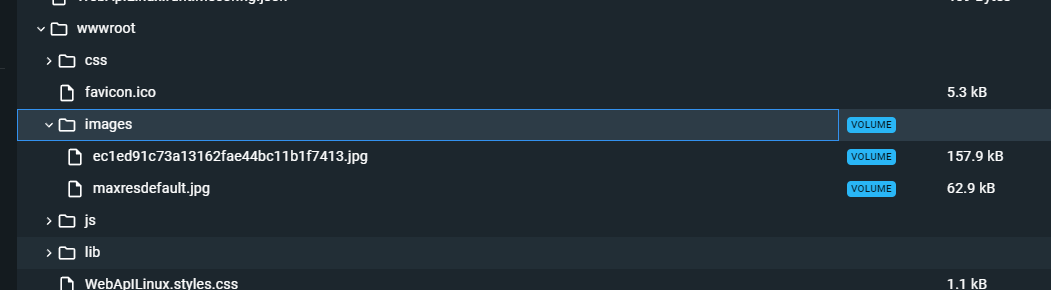


Рисунок 96. Файлы контейнера.

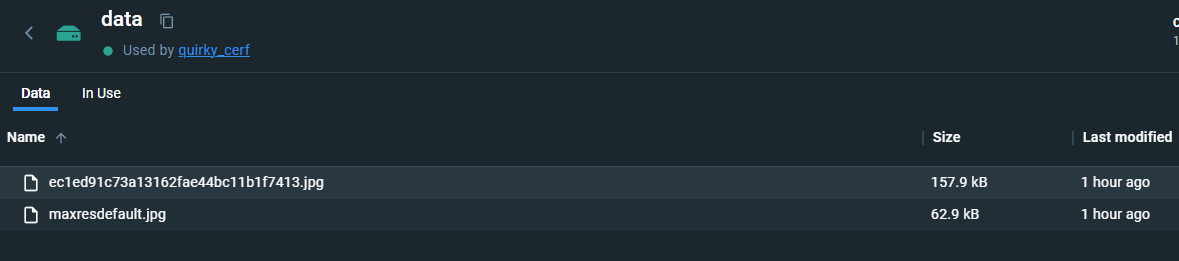


Рисунок 97. Данные тома.

Запускаем новый контейнер с привязкой к папке хоста.



Рисунок 98. Запуск контейнера с прявязкой к папке хоста.

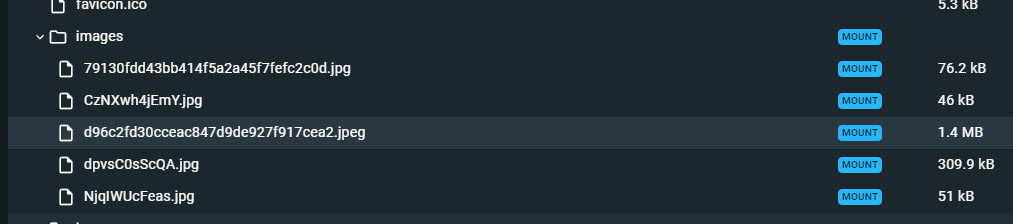


Рисунок 99. Привязанные файлы в контейнере.

Создаём файл python с библиотекой Flask для работы API

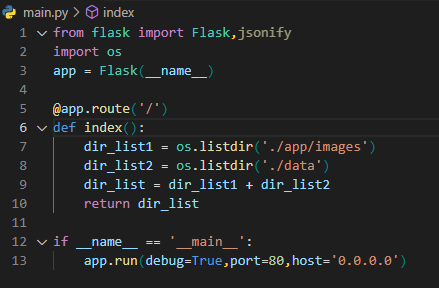


Рисунок 100. Файл api.

Создаём Dockerfile. Указываем название тома и папку. Копируем каритинки и файл в папки. Устанавливаем библиотеку. Указываем порт. Запускаем файл.

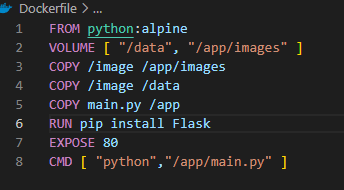


Рисунок 101. Dockerfile к api.

Собираем контейнер и запускаем с привязкой к папке с другими картинками.

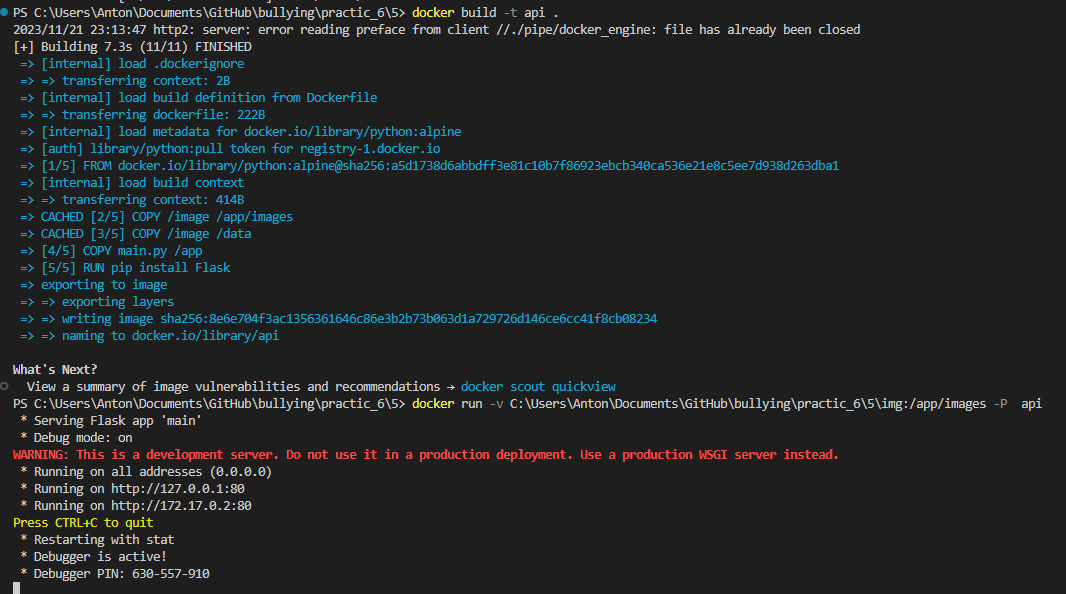


Рисунок 102. Сборка и запуск контейнера.

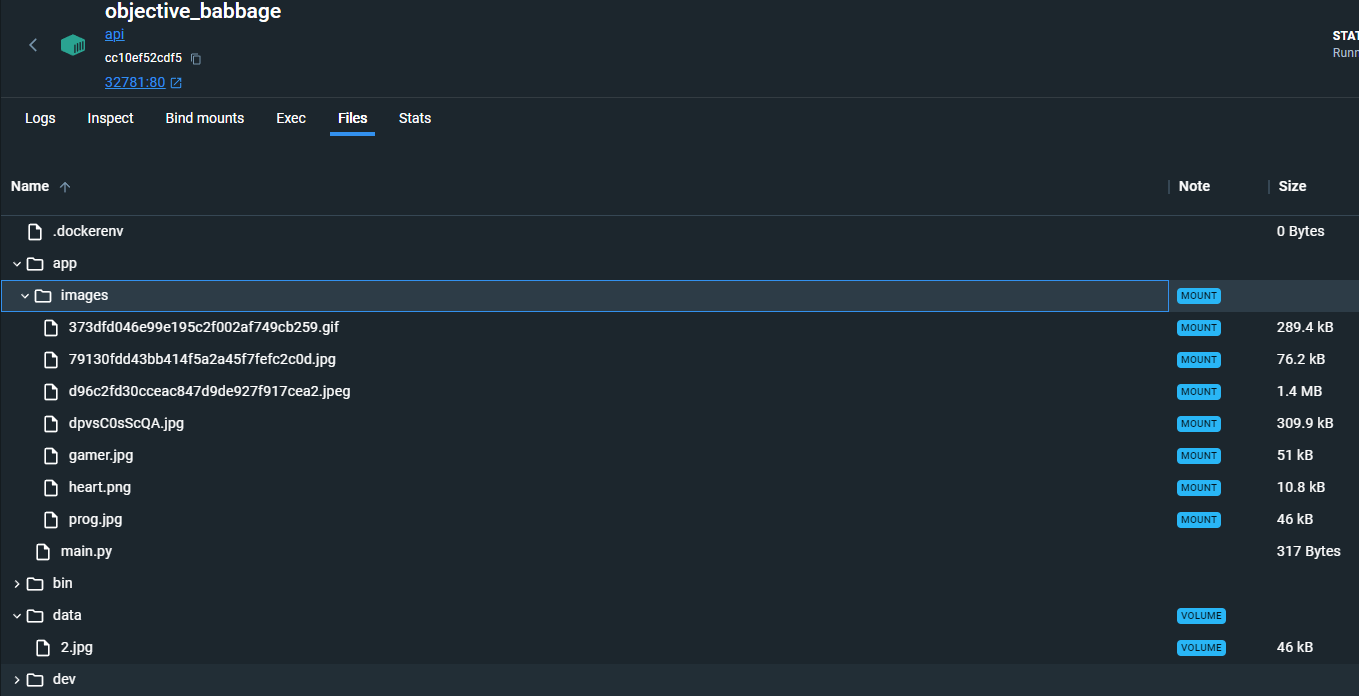


Рисунок 103. Файлы контейнера api.

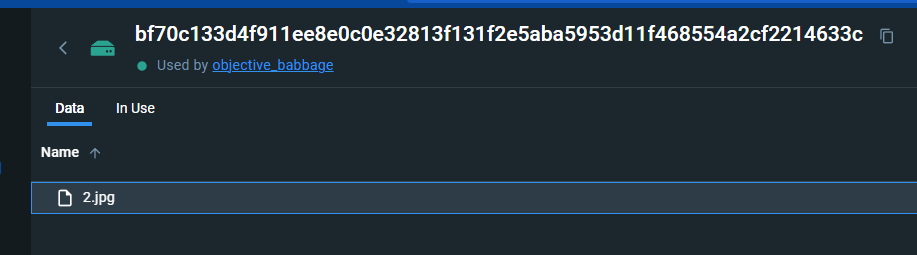


Рисунок 104. Данные тома api.

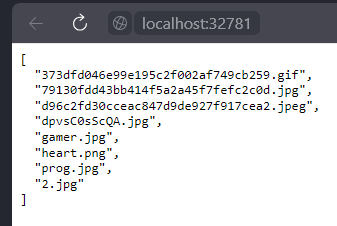


Рисунок 105. Результат.

Вывод: Была выполнена поставленная задача. Были получены новые знания и усвоены.

# Практическая работа №7.

Цель работы:

1. Необходимо развернуть и настроить свой локальный Gitlab и Gitlab Runner.

2. Описать вкладки и функции Admin Area

3. Добавить от лица администратора еще одного пользователя.

4. Авторизоваться в вашем Gitlab под данным пользователем с другого устройства, находящегося в основным сервером Gitlab в одной локальной сети.

5. Создать проект от лица администратора, добавить туда пользователя, и от лица этого пользователя с его устройства запушить изменения на ваш Gitlab.

6. Зарегистрировать в проекте Gitlab Runner

Создание папок C:\gitlab\opt\gitlab\config, C:\gitlab\opt\gitlab\data, C:\gitlab\opt\gitlab\gitlab-runner, C:\gitlab\opt\gitlab\logs, C:\gitlab\opt\gitlab\run\docker.sock, C:\gitlab\opt\gitlab\run и docker-compose

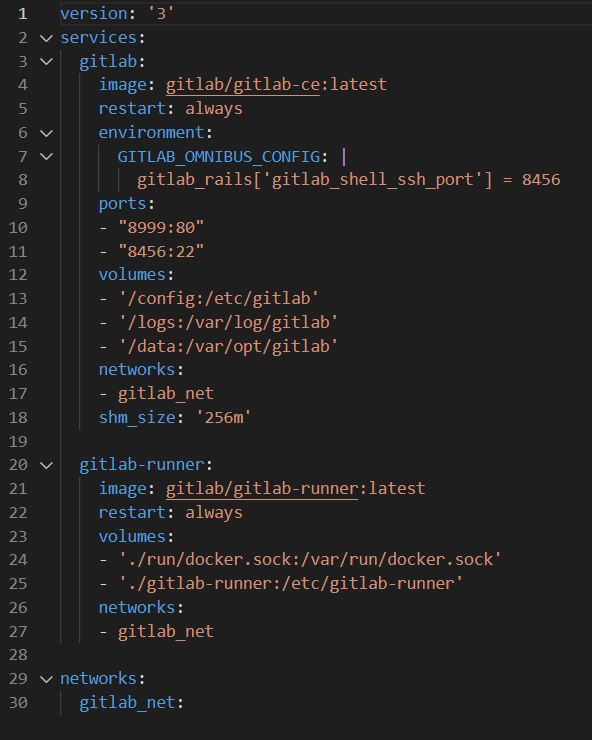


Рисунок 106. Docker-compose gitlab.

Создание контейнеров.



Рисунок 107. Запуск контейнера.

Admin Area:

Project – Общее количество проектов, до 10 последних проектов и возможность создания нового проекта

Users – Общее количество пользователей, до 10 последних пользователей, возможность создания нового пользователя и ссылка на статистику пользователей. Groups – Общее количество групп, до 10 последних групп и возможность создания новой группы

Statistic - Общее количество всех элементов экземпляра GitLab

Features - Все функции, доступные в экземпляре GitLab. Включенные функции отмечены значком зеленого круга, а отключенные - значком включения.

Components – Основные компоненты GitLab и номер версии каждого из них. Также включена ссылка на серверы Gitaly.

Last project – посдная активность с время

Last users - посдная активность с время

Last groups - посдная активность с время

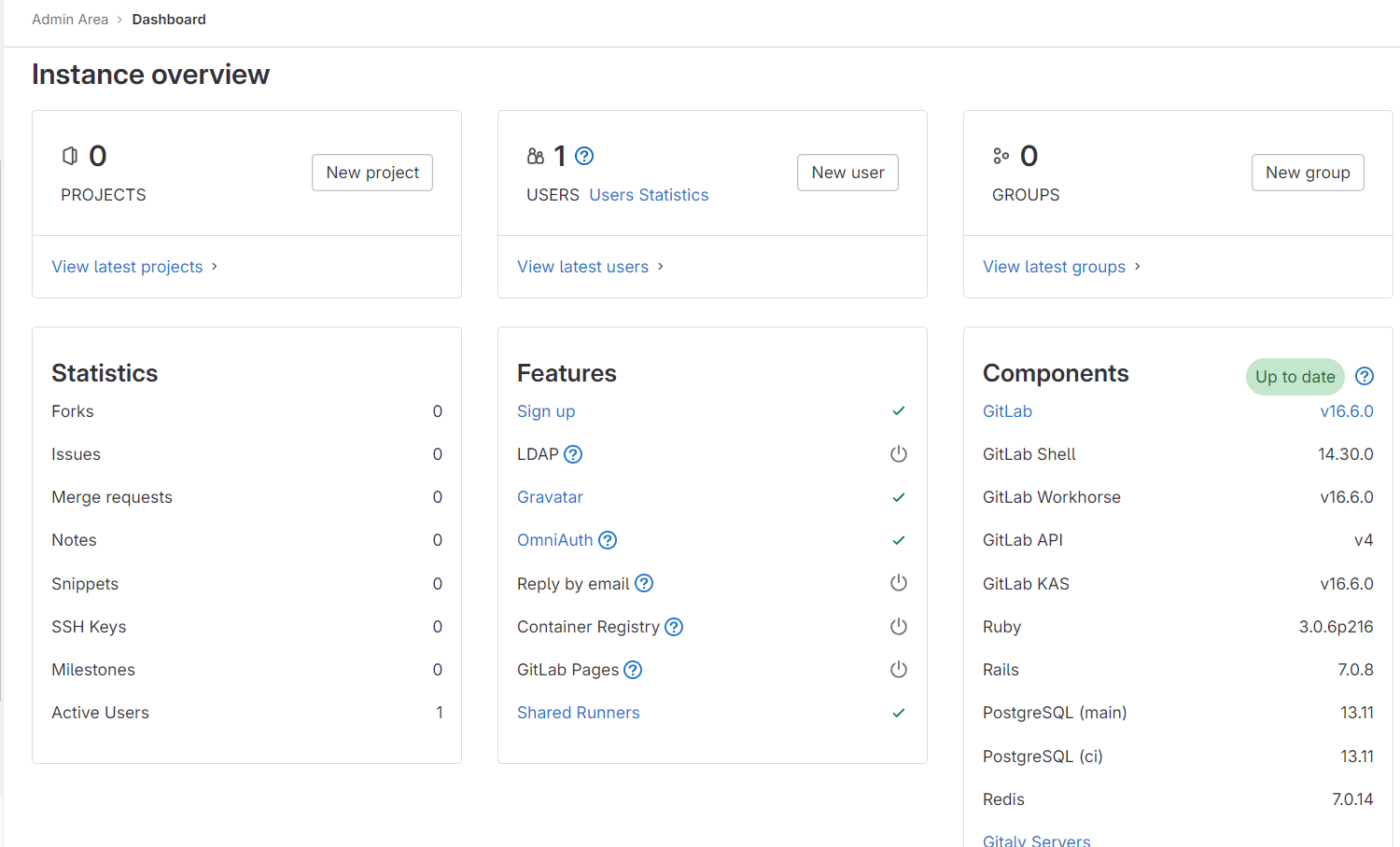


Рисунок 108. Admin Area часть 1

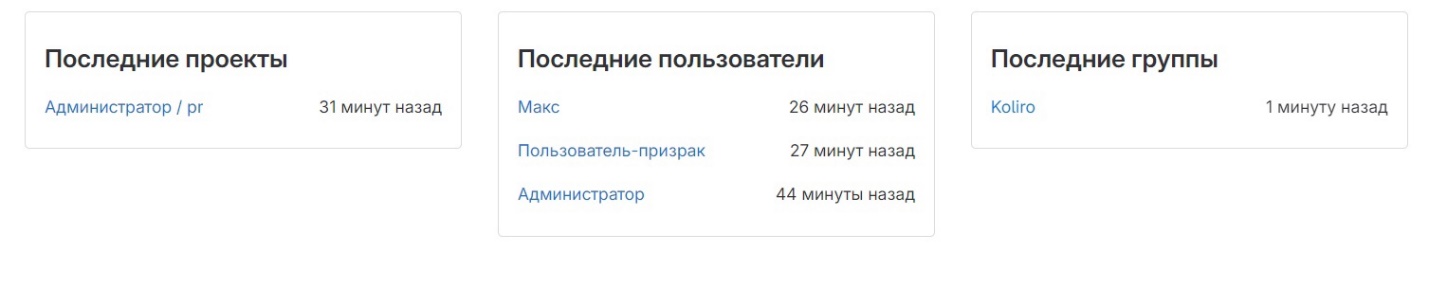


Рисунок 109. Admin Area часть 2

Создание нового пользователя.

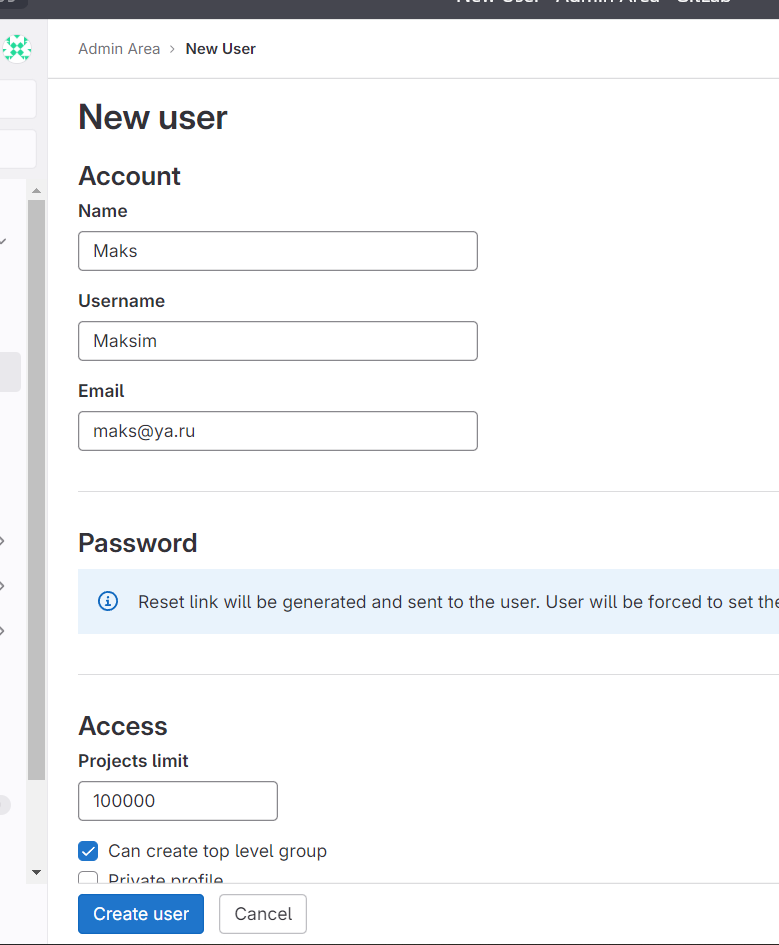


Рисунок 110. Новый пользователь.

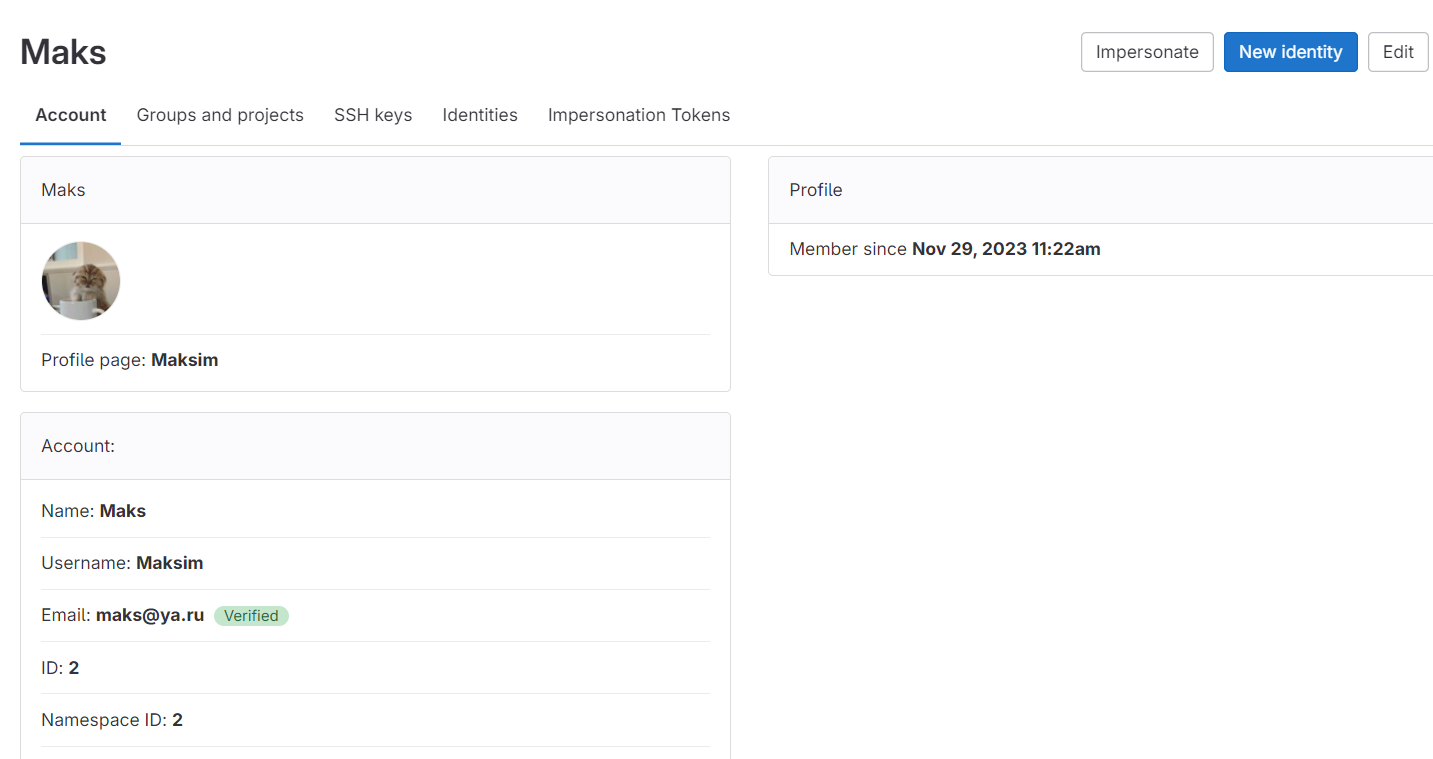


Рисунок 111. Информация о пользователе.

Создание проекта.

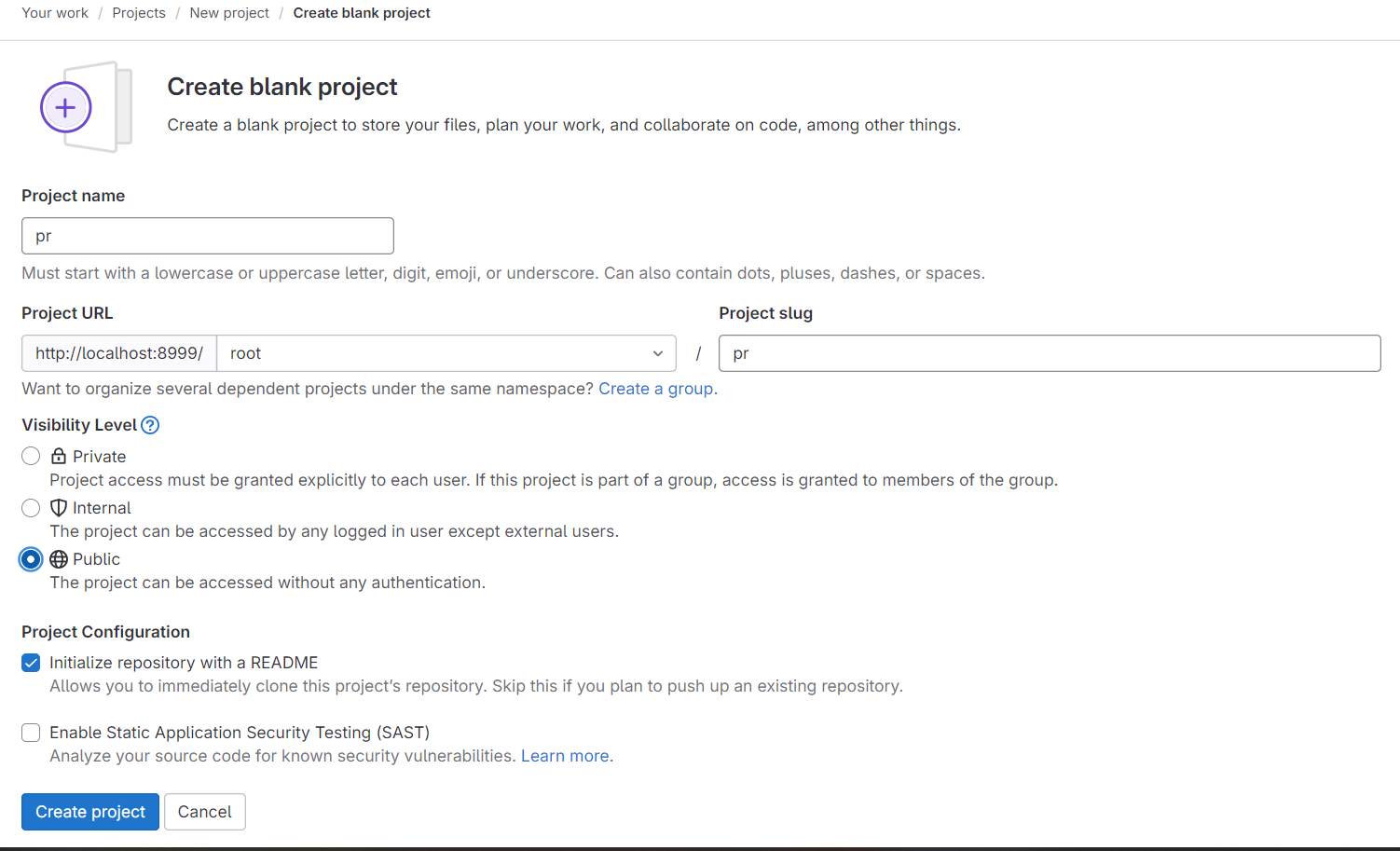


Рисунок 112. Создание проекта.

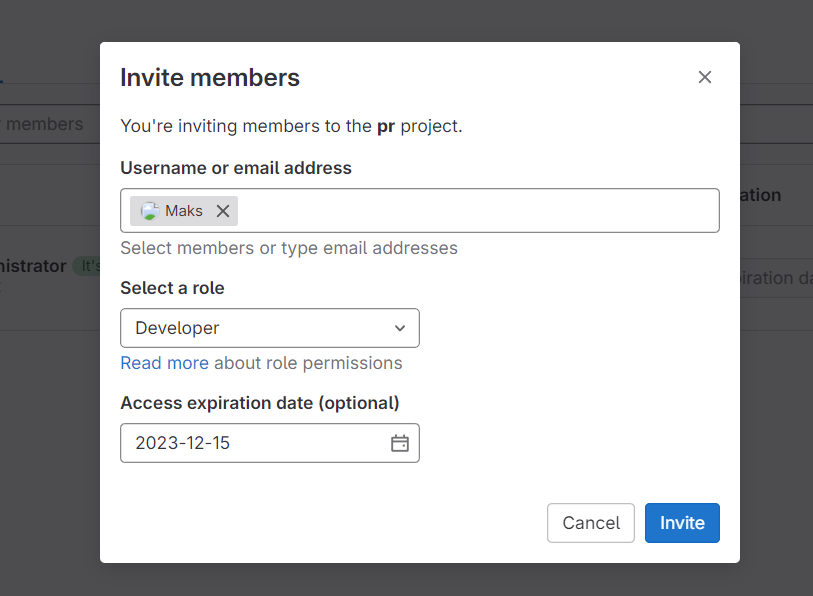


Рисунок 113. Добавление участника в проект под ролью разработчика.

Отправка репозитория с другого локального компьютера.

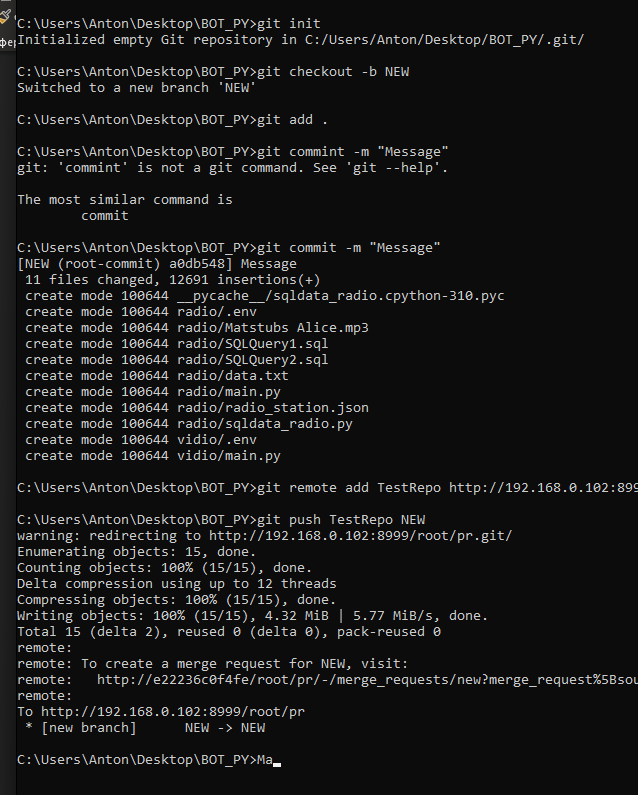


Рисунок 114. CMD второго компьютера.

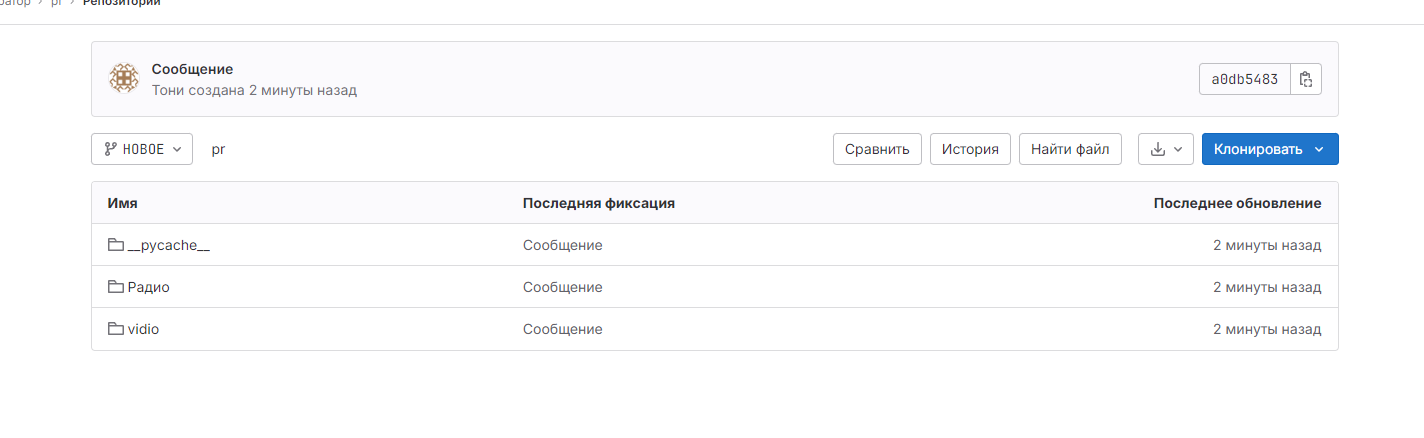


Рисунок 115. Проект в gitlab'е

Добавление runner’а.

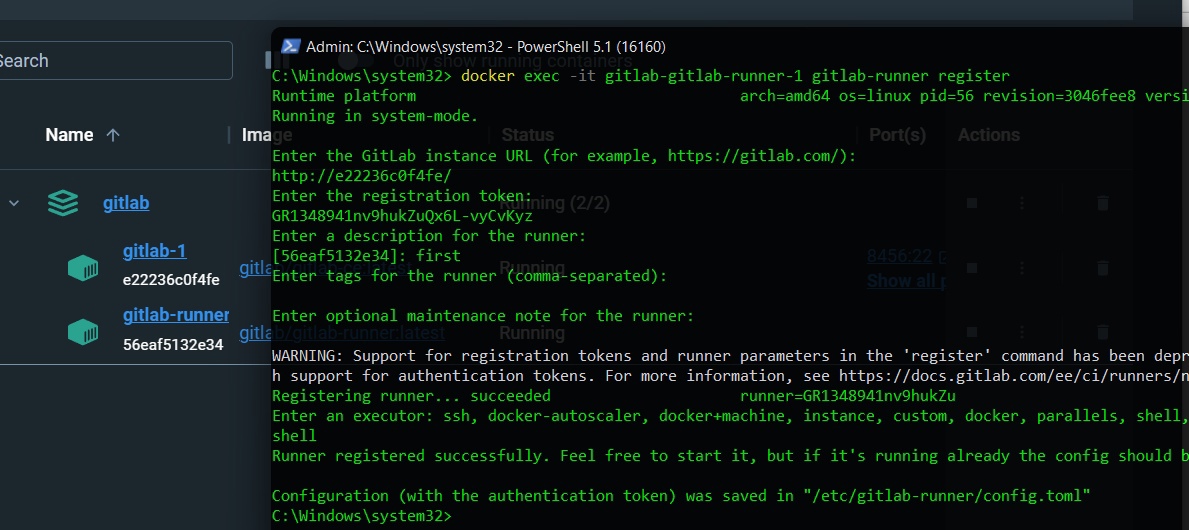


Рисунок 116. Shell runner.

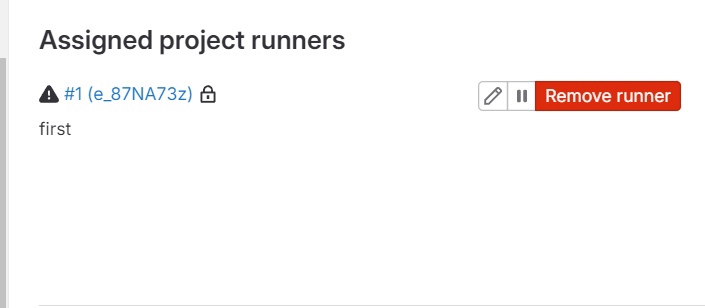


Рисунок 117. Информация.

Вывод: Была выполнена поставленная задача. Были получены новые знания и усвоены.

# Практическая работа №8.

Цель работы: Настройка CI/CD-пайплайна с использованием GitLab CI

1. Лекционный материал

2. Написать пайплайн для CI/CD, который будет иметь как минимум 3 stages, одна из которых функциональна (например сборка приложения), а также они связаны друг с другом

CI/CD представляет собой методологию разработки, которая стремится автоматизировать каждый шаг процесса разработки приложений.

Основная цель CI/CD - упростить и ускорить процесс интеграции кода от разработчиков и его развертывания.

CI/CD помогает повысить качество и надежность приложений, минимизируя возможные ошибки во время интеграции.

Один из основных принципов CI/CD - регулярное и частое интегрирование нового кода для предотвращения конфликтов и сбоев в работе приложения.

CI/CD также предлагает автоматическую сборку и тестирование приложений, чтобы разработчики могли быстро получать обратную связь об их работе.

С помощью CI/CD команды разработчиков могут быстро выявить и исправить ошибки в коде, еще до того, как они попадут в продакшн.

Continuous Integration (непрерывная интеграция) – это практика, заключающаяся в частом интегрировании кода разработчиков в общую базу. При использовании CI команда разработки регулярно коммитит свои изменения в общий репозиторий, а затем система автоматически собирает и проверяет код на наличие ошибок и конфликтов. Это позволяет выявлять проблемы в коде на ранних этапах разработки и снижать время на объединение изменений от разных разработчиков.

Continuous Delivery (непрерывная доставка) - это практика, которая позволяет автоматизировать процесс доставки приложений в производственную среду. При использовании CD каждый успешный билд проходит через набор тестов, автоматическую сборку, а затем деплоится в среду тестирования, предпринимательскую среду и, в конечном счете, приложение деплоится в продакшн в ручном формате. Это позволяет команде разработки быстро и надежно доставлять продукт пользователям.

Continuous Deployment (непрерывное развертывание) – это практика, которая позволяет автоматизировать процесс доставки приложений в производственную среду. При использовании CD каждый успешный билд проходит через набор тестов, автоматическую сборку, а затем деплоится в среду тестирования, предпринимательскую среду и, в конечном счете, приложение деплоится в продакшн в автоматическом режиме, т.е. разворачивается само. Это позволяет команде разработки быстро и надежно доставлять продукт пользователям.

Пайплайн – это последовательность шагов или процессов, которые выполняются для достижения определенного результата.

Одна из основных целей пайплайна - устранение ручной работы и повышение эффективности.

Пайплайн может использоваться в различных отраслях, таких как IT, производство, маркетинг и т.д.

В IT пайплайн может включать этапы разработки, тестирования, развертывания и обновления программного обеспечения.

Для создания пайплайнов в IT можно использовать различные инструменты, такие как Jenkins, GitLab CI/CD, AWS CodePipeline и другие.

GitLab CI - это инструмент для автоматической сборки, тестирования и развертывания программного кода при использовании системы контроля версий GitLab. Он позволяет разработчикам интегрировать свои проекты с процессом непрерывной интеграции и доставки (CI/CD), что позволяет детектировать ошибки и конфликты кода на ранних стадиях разработки. GitLab CI позволяет определить набор шагов и условий для автоматической проверки кода и создания уведомлений о статусе сборок.

.gitlab-ci.yml – это файл конфигурации для системы Continuous Integration/Continuous Deployment (CI/CD) GitLab CI/CD. Он используется для описания и настройки пайплайна автоматической сборки, тестирования и развертывания проекта.

Этот файл хранится в корневом каталоге проекта GitLab и автоматически обрабатывается GitLab CI/CD при изменении репозитория.

before\_script: Команда или серия команд, которые будут выполнены перед запуском задач в остальной части пайплайна.

stages: Список всех этапов (stage), которые будут использоваться в CI/CD-пайплайне

job: Задача, которая будет выполнена в CI/CD-пайплайне.

script: Список команд и скриптов, которые должны быть выполнены в одной задаче.

only/except: Условия, определяющие, когда задача будет запущена или пропущена в пайплайне.

dependencies: Чаще всего употребляется внутри job. Определяет зависимости задачи от других задач. Например, чтобы выполнить задачу deploy, необходимо предварительно выполнить задачу build.

needs: Также употребляется внутри job. Определяет, что задача зависит от другой задачи и должна быть запущена только после ее успешного выполнения.

when: определяет условия выполнения задач.

Значения:

on\_success: Задача будет выполняться только при успешном выполнении предыдущих задач.

on\_failure: Задача будет выполняться только в случае ошибки или сбоя в предыдущих задачах.

always: Задача всегда будет выполняться, независимо от результата предыдущих задач.

manual: Задача требует ручного запуска через интерфейс GitLab.

Создаём файл.



Рисунок 118. файл gitlab-ci.yml

Пишем 3 этапа: Тест, Деплой, Релиз. Указываем что выплняем следущие этапы при успешном выполнении прошлой. Указываем действия в script. Указываем что делать после исполнения скрипта before\_script. Сначала указываем название а потом в stage указываем к какому этапу он относится.

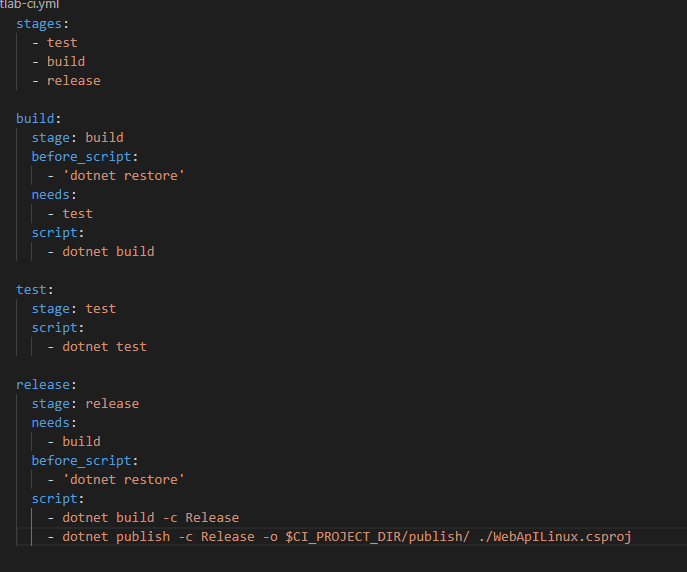


Рисунок 119. код gitlab-ci.yml



Рисунок 120. Запуск CI/CD

Вывод: Была выполнена поставленная задача. Были получены новые знания и усвоены.

# Дополнительное задание.

Цель задания: развернуть любое WPF приложение внутри докера. Это может быть даже просто приложение с кнопкой, которая переключает на другое окно. Сам состав приложения значения не имеет.

Создаём Dockerfile где указываем что надо скачать: sudo, wget, software-properties-common и gpg.

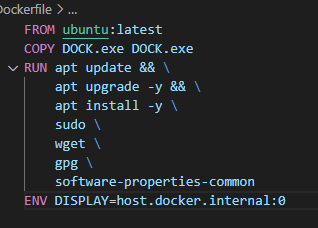


Рисунок 121.Код образа

После создания файла в терминале прописываем команды по порядку. В вспылаещем окне нажать кнопку «Install».

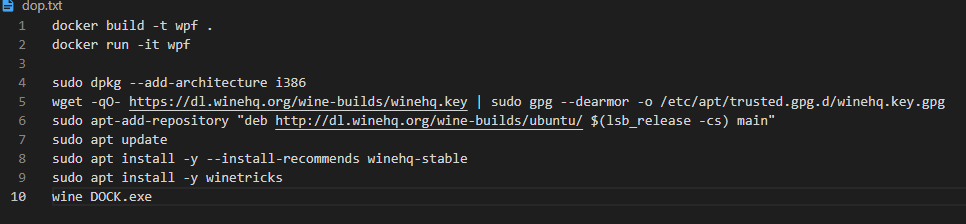


Рисунок 122. Команды

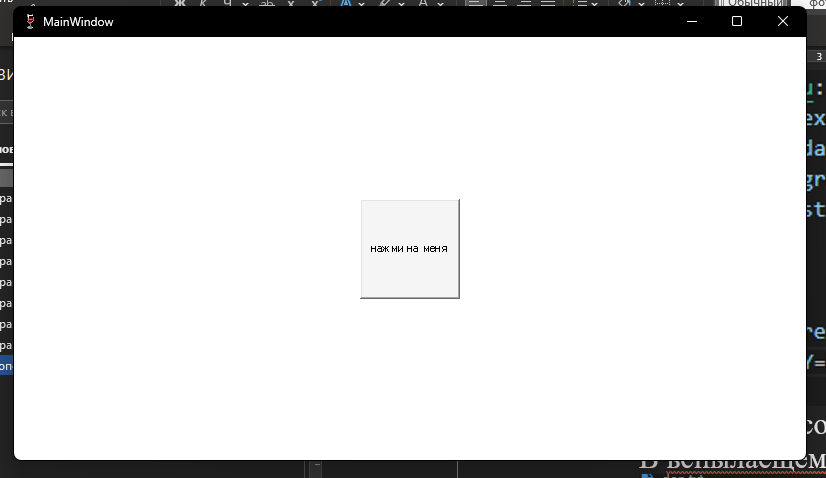


Рисунок 123. Результат 1.

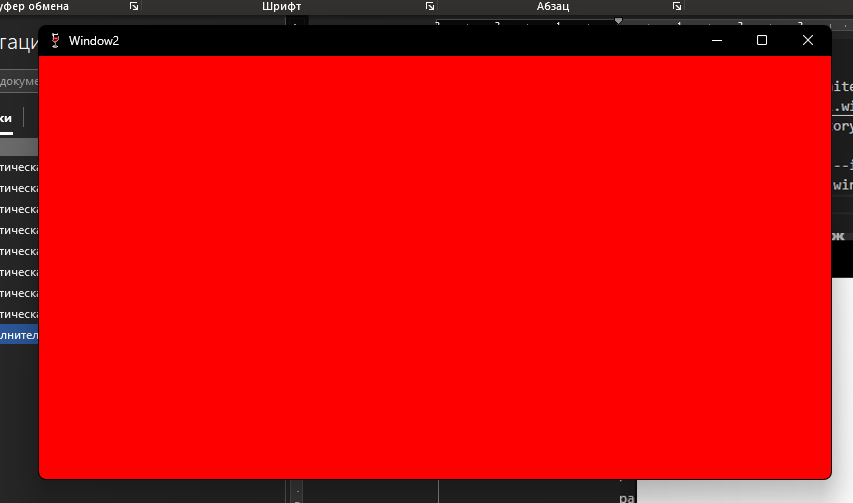


Рисунок 124. Результат 2.

Вывод: Была выполнена поставленная задача. Были получены новые знания и усвоены.