Отчёт по лабораторной работе №4

Продвинутое использование git.

Седохин Даниил Алексеевич

Содержание

# 1 Цель работы

Получение навыков правильной работы с репозиториями git.

# 2 Задание

Выполнить работу для тестового репозитория.   
Преобразовать рабочий репозиторий в репозиторий с git-flow и conventional commits.

# 3 Теоретическое введение

Рабочий процесс Gitflow

Рабочий процесс Gitflow Workflow. Будем описывать его с использованием пакета git-flow.

Общая информация

Gitflow Workflow опубликована и популяризована Винсентом Дриссеном.   
Gitflow Workflow предполагает выстраивание строгой модели ветвления с учётом выпуска проекта.   
Данная модель отлично подходит для организации рабочего процесса на основе релизов.   
Работа по модели Gitflow включает создание отдельной ветки для исправлений ошибок в рабочей среде.   
Последовательность действий при работе по модели Gitflow:   
 Из ветки master создаётся ветка develop.   
 Из ветки develop создаётся ветка release.   
 Из ветки develop создаются ветки feature.   
 Когда работа над веткой feature завершена, она сливается с веткой develop.   
 Когда работа над веткой релиза release завершена, она сливается в ветки develop и master.   
 Если в master обнаружена проблема, из master создаётся ветка hotfix.   
 Когда работа над веткой исправления hotfix завершена, она сливается в ветки develop и master.

Процесс работы с Gitflow

Основные ветки (master) и ветки разработки (develop)   
 Для фиксации истории проекта в рамках этого процесса вместо одной ветки master используются две ветки. В ветке master хранится официальная история релиза, а ветка develop предназначена для объединения всех функций. Кроме того, для удобства рекомендуется присваивать всем коммитам в ветке master номер версии.  
  
 При использовании библиотеки расширений git-flow нужно инициализировать структуру в существующем репозитории:   
  
 git flow init   
  
 Для github параметр Version tag prefix следует установить в v.   
  
 После этого проверьте, на какой ветке Вы находитесь:   
  
 git branch   
  
Функциональные ветки (feature)   
 Под каждую новую функцию должна быть отведена собственная ветка, которую можно отправлять в центральный репозиторий для создания резервной копии или совместной работы команды. Ветки feature создаются не на основе master, а на основе develop. Когда работа над функцией завершается, соответствующая ветка сливается обратно с веткой develop. Функции не следует отправлять напрямую в ветку master.  
 Как правило, ветки feature создаются на основе последней ветки develop.  
  
 Создание функциональной ветки  
  
 Создадим новую функциональную ветку:  
  
 git flow feature start feature\_branch  
  
 Далее работаем как обычно.  
  
 Окончание работы с функциональной веткой  
  
 По завершении работы над функцией следует объединить ветку feature\_branch с develop:  
  
 git flow feature finish feature\_branch  
  
Ветки выпуска (release)  
 Когда в ветке develop оказывается достаточно функций для выпуска, из ветки develop создаётся ветка release. Создание этой ветки запускает следующий цикл выпуска, и с этого момента новые функции добавить больше нельзя — допускается лишь отладка, создание документации и решение других задач. Когда подготовка релиза завершается, ветка release сливается с master и ей присваивается номер версии. После нужно выполнить слияние с веткой develop, в которой с момента создания ветки релиза могли возникнуть изменения.  
 Благодаря тому, что для подготовки выпусков используется специальная ветка, одна команда может дорабатывать текущий выпуск, в то время как другая команда продолжает работу над функциями для следующего.  
  
 Создать новую ветку release можно с помощью следующей команды:  
  
 git flow release start 1.0.0  
  
 Для завершения работы на ветке release используются следующие команды:  
  
 git flow release finish 1.0.0  
  
Ветки исправления (hotfix)  
 Ветки поддержки или ветки hotfix используются для быстрого внесения исправлений в рабочие релизы. Они создаются от ветки master. Это единственная ветка, которая должна быть создана непосредственно от master. Как только исправление завершено, ветку следует объединить с master и develop. Ветка master должна быть помечена обновлённым номером версии.  
 Наличие специальной ветки для исправления ошибок позволяет команде решать проблемы, не прерывая остальную часть рабочего процесса и не ожидая следующего цикла релиза.  
  
 Ветку hotfix можно создать с помощью следующих команд:  
  
 git flow hotfix start hotfix\_branch  
  
 По завершении работы ветка hotfix объединяется с master и develop:  
  
 git flow hotfix finish hotfix\_branch

Семантическое версионирование

Семантический подход в версионированию программного обеспечения.

Краткое описание семантического версионирования

Семантическое версионирование описывается в манифесте семантического версионирования.  
  
Кратко его можно описать следующим образом:  
 Версия задаётся в виде кортежа МАЖОРНАЯ\_ВЕРСИЯ.МИНОРНАЯ\_ВЕРСИЯ.ПАТЧ.  
 Номер версии следует увеличивать:  
 МАЖОРНУЮ версию, когда сделаны обратно несовместимые изменения API.  
 МИНОРНУЮ версию, когда вы добавляете новую функциональность, не нарушая обратной совместимости.  
 ПАТЧ-версию, когда вы делаете обратно совместимые исправления.  
 Дополнительные обозначения для предрелизных и билд-метаданных возможны как дополнения к МАЖОРНАЯ.МИНОРНАЯ.ПАТЧ формату.

Программное обеспечение

Для реализации семантического версионирования создано несколько программных продуктов.  
При этом лучше всего использовать комплексные продукты, которые используют информацию из коммитов системы версионирования.  
Коммиты должны иметь стандартизованный вид.  
В семантическое версионирование применяется вместе с общепринятыми коммитами.  
  
Пакет Conventional Changelog  
 Пакет Conventional Changelog является комплексным решением по управлению коммитами и генерации журнала изменений.  
 Содержит набор утилит, которые можно использовать по-отдельности.

Общепринятые коммиты

Использование спецификации Conventional Commits.

Описание

Спецификация Conventional Commits:

Соглашение о том, как нужно писать сообщения commit'ов.  
Совместимо с SemVer. Даже вернее сказать, сильно связано с семантическим версионированием.  
Регламентирует структуру и основные типы коммитов.  
  
Структура коммита  
  
<type>(<scope>): <subject>   
<BLANK LINE>   
<body>   
<BLANK LINE>   
<footer>   
  
Или, по-русски:   
  
<тип>(<область>): <описание изменения>   
<пустая линия>   
[необязательное тело]   
<пустая линия>   
[необязательный нижний колонтитул]   
  
 Заголовок является обязательным.  
 Любая строка сообщения о фиксации не может быть длиннее 100 символов.  
 Тема (subject) содержит краткое описание изменения.  
 Используйте повелительное наклонение в настоящем времени: «изменить» ("change" not "changed" nor "changes").  
 Не используйте заглавную первую букву.  
 Не ставьте точку в конце.  
 Тело (body) должно включать мотивацию к изменению и противопоставлять это предыдущему поведению.  
 Как и в теме, используйте повелительное наклонение в настоящем времени.  
 Нижний колонтитул (footer) должен содержать любую информацию о критических изменениях.  
 Следует использовать для указания внешних ссылок, контекста коммита или другой мета информации.  
 Также содержит ссылку на issue (например, на github), который закрывает эта фиксация.  
 Критические изменения должны начинаться со слова BREAKING CHANGE: с пробела или двух символов новой строки. Затем для этого используется остальная часть сообщения фиксации.  
  
Типы коммитов  
  
 Базовые типы коммитов  
 fix: — коммит типа fix исправляет ошибку (bug) в вашем коде (он соответствует PATCH в SemVer).  
 feat: — коммит типа feat добавляет новую функцию (feature) в ваш код (он соответствует MINOR в SemVer).  
 BREAKING CHANGE: — коммит, который содержит текст BREAKING CHANGE: в начале своего не обязательного тела сообщения (body) или в подвале (footer), добавляет изменения, нарушающие обратную совместимость вашего API (он соответствует MAJOR в SemVer). BREAKING CHANGE может быть частью коммита любого типа.  
 revert: — если фиксация отменяет предыдущую фиксацию. Начинается с revert:, за которым следует заголовок отменённой фиксации. В теле должно быть написано: Это отменяет фиксацию <hash> (это SHA-хэш отменяемой фиксации).  
 Другое: коммиты с типами, которые отличаются от fix: и feat:, также разрешены. Например, @commitlint/config-conventional (основанный на The Angular convention) рекомендует: chore:, docs:, style:, refactor:, perf:, test:, и другие.  
  
 Соглашения The Angular convention  
 Одно из популярных соглашений о поддержке исходных кодов — конвенция Angular (The Angular convention).  
  
 Типы коммитов The Angular convention  
  
 Конвенция Angular (The Angular convention) требует следующие типы коммитов:  
 build: — изменения, влияющие на систему сборки или внешние зависимости (примеры областей (scope): gulp, broccoli, npm).  
 ci: — изменения в файлах конфигурации и скриптах CI (примеры областей: Travis, Circle, BrowserStack, SauceLabs).  
 docs: — изменения только в документации.  
 feat: — новая функция.  
 fix: — исправление ошибок.  
 perf: — изменение кода, улучшающее производительность.  
 refactor: — Изменение кода, которое не исправляет ошибку и не добавляет функции (рефакторинг кода).  
 style: — изменения, не влияющие на смысл кода (пробелы, форматирование, отсутствие точек с запятой и т. д.).  
 test: — добавление недостающих тестов или исправление существующих тестов.  
  
 Области действия (scope)  
  
 Областью действия должно быть имя затронутого пакета npm (как его воспринимает человек, читающий журнал изменений, созданный из сообщений фиксации).  
  
 Есть несколько исключений из правила «использовать имя пакета»:  
 packaging — используется для изменений, которые изменяют структуру пакета, например, изменения общедоступного пути.  
 changelog — используется для обновления примечаний к выпуску в CHANGELOG.md.  
 отсутствует область действия — полезно для изменений стиля, тестирования и рефакторинга, которые выполняются во всех пакетах (например, style: добавить отсутствующие точки с запятой).  
  
 Соглашения @commitlint/config-conventional  
  
 Соглашение @commitlint/config-conventional входит в пакет Conventional Changelog. В целом в этом соглашении придерживаются соглашения Angular.

# 4 Выполнение лабораторной работы

1. Установим git-flow:

* dnf copr enable elegos/gitflow  
  dnf install gitflow (рис. 1 2).

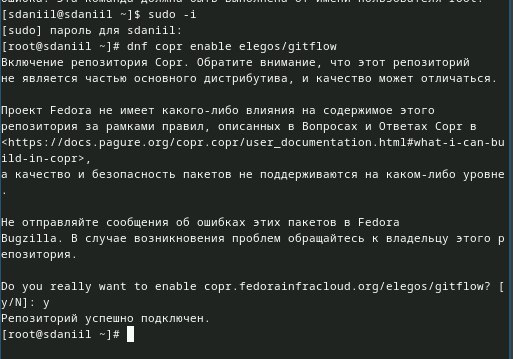


Рис. 1: Установка git-flow

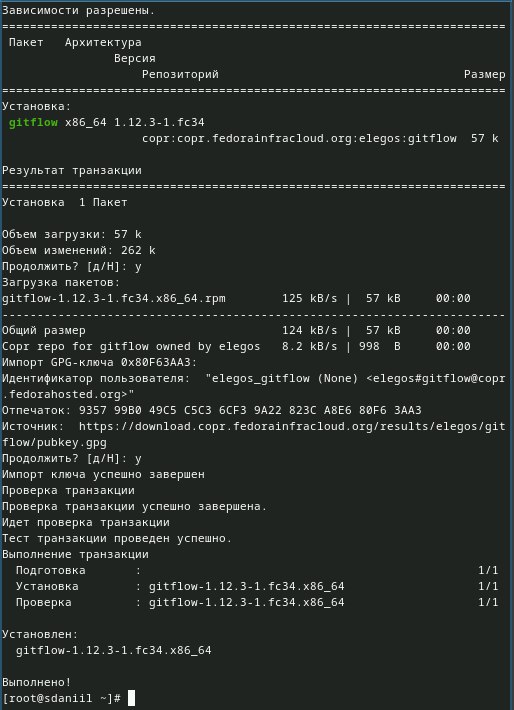


Рис. 2: Установка git-flow

1. Установка Node.js

На Node.js базируется программное обеспечение для семантического версионирования и общепринятых коммитов.

dnf install nodejs   
apt-get install pnpm. (рис. [-@fig:003]).

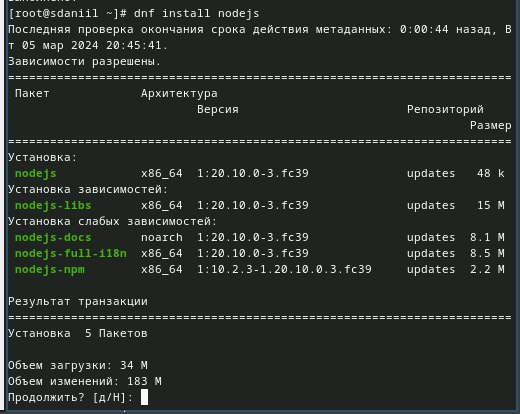


Рис. 3: Установка Node.js

1. Настройка Node.js

Для работы с Node.js добавим каталог с исполняемыми файлами, устанавливаемыми yarn, в переменную PATH.

Запустим:   
  
pnpm setup   
  
Перелогинимся, или выполним:   
  
source ~/.bashrc. (рис. [-@fig:004]).

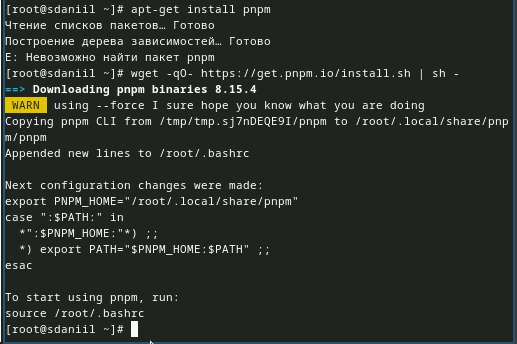


Рис. 4: Запуск pnpm

1. commitizen

* Данная программа используется для помощи в форматировании коммитов.
* pnpm add -g commitizen
* При этом устанавливается скрипт git-cz, который мы и будем использовать для коммитов. (рис. 5)

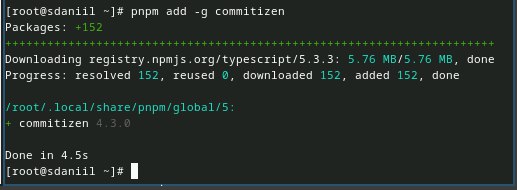


Рис. 5: Установка скрипта git-cz

1. standard-changelog

* Данная программа используется для помощи в создании логов.
* pnpm add -g standard-changelog (рис. 6).

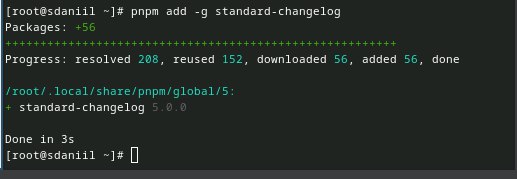


Рис. 6: Имя начальной ветки и параметры

1. Практический сценарий использования git

* Создание репозитория git  
  Создадим репозиторий на GitHub. Для примера назовём его git-extended. (рис. 7).

Репозиторий git-extended

Рис. 7: Репозиторий git-extended

1. Делаем первый коммит и выкладываем на github:

git commit -m “first commit”  
git remote add origin git@github.com:/git-extended.git  
git push -u origin master (рис. 8).

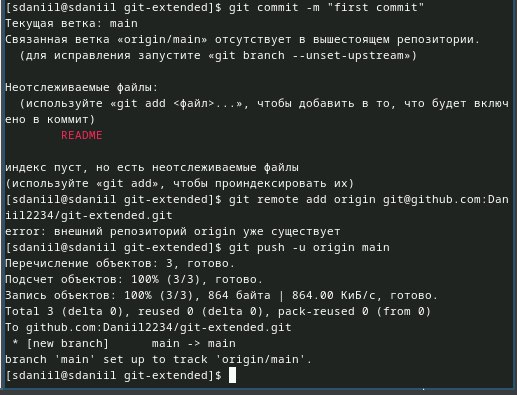


Рис. 8: Коммит

1. Конфигурация для пакетов Node.js

pnpm init

Необходимо заполнить несколько параметров пакета.

Название пакета.   
Лицензия пакета.   
 Список лицензий для npm: https://spdx.org/licenses/. Предлагается выбирать лицензию CC-BY-4.0. (рис. [-@fig:0010]).

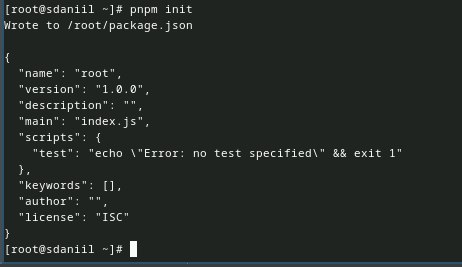


Рис. 9: Конфигурация для пакетов Jode.js

1. Сконфигурим формат коммитов. Для этого добавим в файл package.json команду для формирования коммитов:  
   “config”: {  
   “commitizen”: {  
   “path”: “cz-conventional-changelog”  
   }  
   }  
   Таким образом, файл package.json приобретает вид: (рис. 10).

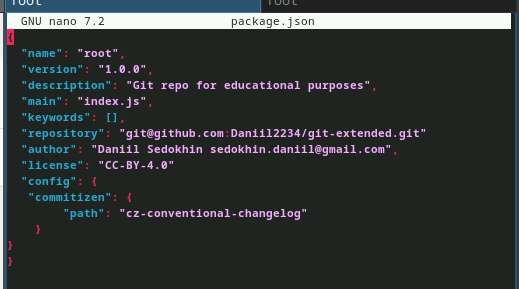


Рис. 10: Изменение package.json

1. Добавим новые файлы:

git add .

Выполним коммит:

git cz

Отправим на github:

git push (рис. 11).

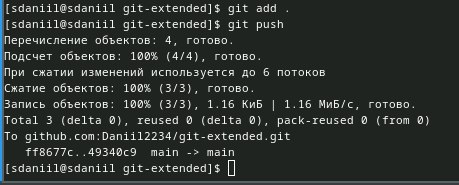


Рис. 11: Добавление новых файлов, коммит и отправка на github

1. Конфигурация git-flow

* Инициализируем git-flow
* git flow init (рис. 12).

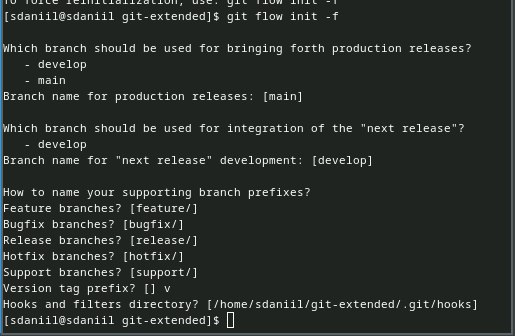


Рис. 12: Инициализация git-flow

1. Префикс для ярлыков установим в v.

Проверьте, что Вы на ветке develop:

git branch

Загрузите весь репозиторий в хранилище:

git push –all (рис. 13).

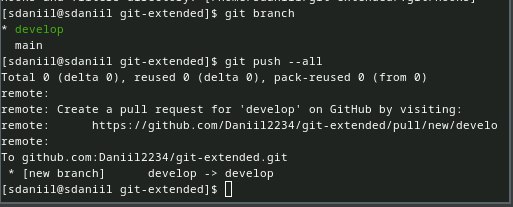


Рис. 13: Загрузка всего репозитория в хранилище

1. Установите внешнюю ветку как вышестоящую для этой ветки:

git branch –set-upstream-to=origin/develop develop (рис. 14).

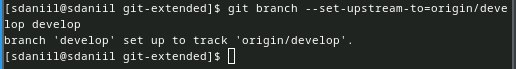


Рис. 14: Установка внешнй ветки как вышестоящую

1. Создадим релиз с версией 1.0.0

git flow release start 1.0.0 (рис. 15).

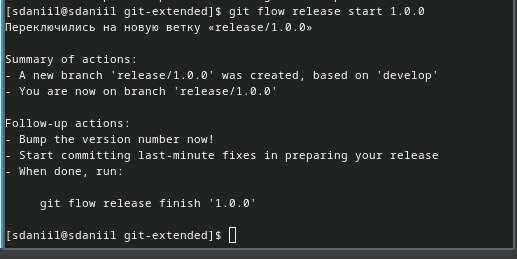


Рис. 15: Создание релиза

1. Создадим журнал изменений

standard-changelog –first-release

Добавим журнал изменений в индекс

git add CHANGELOG.md  
git commit -am ‘chore(site): add changelog’  
Зальём релизную ветку в основную ветку

git flow release finish 1.0.0 (рис. 16).



Рис. 16: Создание журнала изменений и добавление индекса и релиз

1. Отправим данные на github

git push –all  
git push –tags (рис. 17).

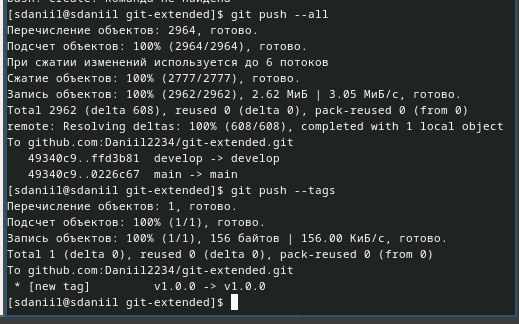


Рис. 17: Отправка файлов на github

1. Создадим релиз на github. Для этого будем использовать утилиты работы с github:

gh release create v1.0.0 -F CHANGELOG.md (рис. 18).

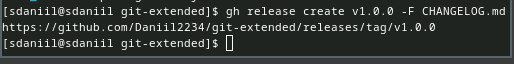


Рис. 18: Создание релиза на github

1. Разработка новой функциональности

* Создадим ветку для новой функциональности:
* git flow feature start feature\_branch (рис. 19).

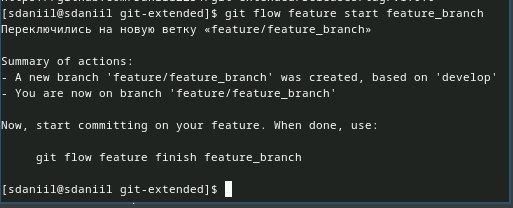


Рис. 19: Создание новой ветки

1. Далее, продолжаем работу c git как обычно.

По окончании разработки новой функциональности следующим шагом следует объединить ветку feature\_branch c develop:

git flow feature finish feature\_branch (рис. 20).

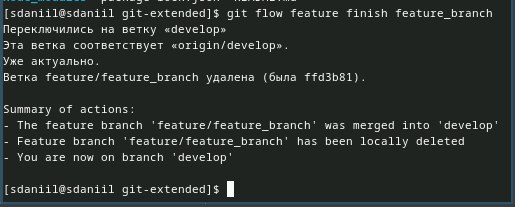


Рис. 20: Объединение ветки

1. Создание релиза git-flow

* Создадим релиз с версией 1.2.3:
* git flow release start 1.2.3 (рис. 21).

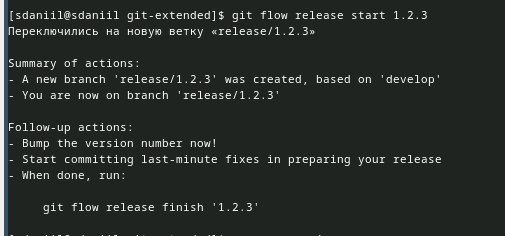
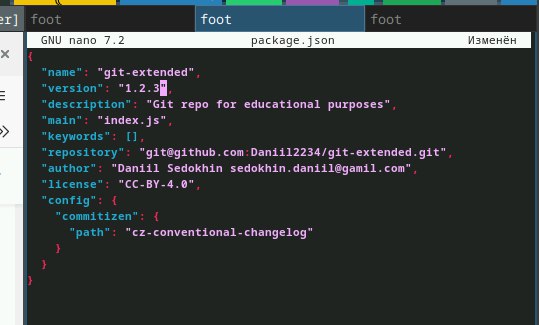


Рис. 21: Создание релиза с версией 1.2.3

1. Обновим номер версии в файле package.json. Установите её в 1.2.3.

* Создадим журнал изменений
* standard-changelog (рис. 22).
* 
* Рис. 22: Редактирование файла package.json

1. Добавим журнал изменений в индекс

git add CHANGELOG.md  
git commit -am ’chore(site): update changel  
Зальём релизную ветку в основную ветку

git flow release finish 1.2.3 (рис. 23).

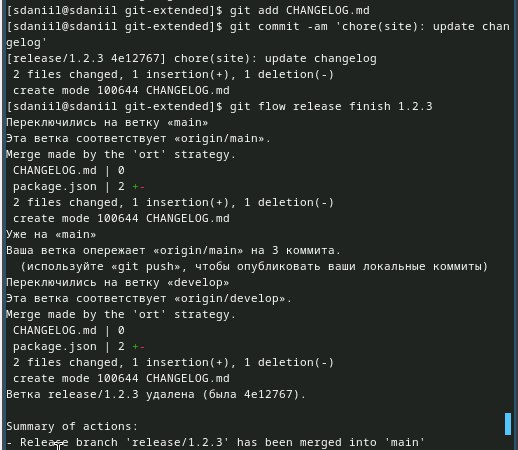


Рис. 23: Добавление журнала изменений в индекс

1. Отправим данные на github

git push –all  
git push –tags

Создадим релиз на github с комментарием из журнала изменений:

gh release create v1.2.3 -F CHANGELOG.md (рис. 24).

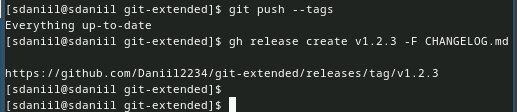


Рис. 24: Отправка данных на github

# 5 Выводы

Я Получил навыки правильной работы с репозиториями git.