

Шаблон отчёта по лабораторной работе

N4

Gitflow

Кхари Жекка Кализая Арсе

Содержание

| | |
|-----------------------------------------------------------------|-----------|
| 1 Цель работы | 5 |
| 2 Задание | 6 |
| 3 Теоретическое введение | 7 |
| 3.1 рабочий процесс Gitflow | 7 |
| 3.1.1 общая информация | 7 |
| 3.1.2 Процесс работы с Gitflow | 8 |
| 3.2 Семантическое версионирование | 10 |
| 3.2.1 Краткое описание семантического версионирования | 10 |
| 3.2.2 Программное обеспечение | 11 |
| 3.3 Общепринятые коммиты | 11 |
| 3.3.1 Описание | 11 |
| 4 Выполнение лабораторной работы | 16 |
| 5 Выводы | 35 |
| Список литературы | 36 |

Список иллюстраций

| | |
|-----------------------------------------------|----|
| 4.1 установка gitflow | 16 |
| 4.2 установка gitflow | 17 |
| 4.3 установка nodejs | 17 |
| 4.4 установка pnpm | 18 |
| 4.5 настройка pnpm | 18 |
| 4.6 активирование gitflow | 19 |
| 4.7 установка пакетов | 19 |
| 4.8 установка пакетов | 20 |
| 4.9 создание репозитория | 21 |
| 4.10 копирование репозитория | 21 |
| 4.11 первый коммит | 22 |
| 4.12 загрузка файлов | 22 |
| 4.13 настройка пакета node.js | 23 |
| 4.14 модификация файла package.json | 24 |
| 4.15 загрузка файлов | 25 |
| 4.16 настройка gitflow | 25 |
| 4.17 проверка ветка | 26 |
| 4.18 загрузка файлов | 26 |
| 4.19 установка ветки | 27 |
| 4.20 релиз версии 1.0.0 | 27 |
| 4.21 модификация журнала | 28 |
| 4.22 добавление журнала | 28 |
| 4.23 нагрузка релизной ветки | 29 |
| 4.24 загрузка на сервер | 29 |
| 4.25 создание релиза | 30 |
| 4.26 Новая ветка | 30 |
| 4.27 работа в ветке | 31 |
| 4.28 создание релиза 1.2.3 | 31 |
| 4.29 изменение файла package.json | 32 |
| 4.30 добавление изменения в журнале | 32 |
| 4.31 нагрузка релизной ветки | 33 |
| 4.32 загрузка файлов на github | 33 |
| 4.33 релиз на github | 34 |

Список таблиц

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

3.1 рабочий процесс Gitflow

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

3.1.1 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`.

3.1.2 Процесс работы с Gitflow

1. Основные ветки (`master`) и ветки разработки (`develop`)

- для фиксации истории проекта в рамках этого процесса вместо доной ветки `master` используются две ветки. `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
```

2. 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`.

1. Создание функциональной ветки

- Создадим новую функциональную ветку:

```
git flow feature start feature_branch
```

- Далее работаем как обычно.

2. Окончание работы с функциональной веткой

- По завершении работы над функцией следует объединить ветку feature_branch с develop:

```
git flow feature finish feature_branch
```

3. 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 сливаются 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
```

4. 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
```

3.2 Семантическое версионирование

Семантический подход в версионированию программного обеспечения

3.2.1 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.

3.2.2 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.

1. Пакет Conventional Changelog

- Пакет Conventional Changelog является комплексным решением по управлению коммитами и генерации журнала изменений.
- Содержит набор утилит, которые можно использовать по-отдельности.

3.3 Общепринятые коммиты

Использование спецификации Conventional Commits.

3.3.1 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.

1. 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.

2. Типы коммитов

1. 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.

2. Соглашения The Angular convention

- Одно из популярных соглашений о поддержке исходных кодов — конвенция Angular (The Angular convention).

1. 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.

2. Области действия (scope)

Областью действия должно быть имя затронутого пакета прм (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).

3. Соглашения @commitlint/config-conventional

Соглашение @commitlint/config-conventional входит в пакет Conventional Changelog. В целом в этом соглашении придерживаются соглашения Angular.

Более подробно про Unix см. в [1–4].

4 Выполнение лабораторной работы

Я начинал эту лабораторную работу, активировав elegos/gitflow и устанавливая gitflow (рис. 4.1) и (рис. 4.2).

```
dnf copr enable elegos/gitflow  
dnf install gitflow
```

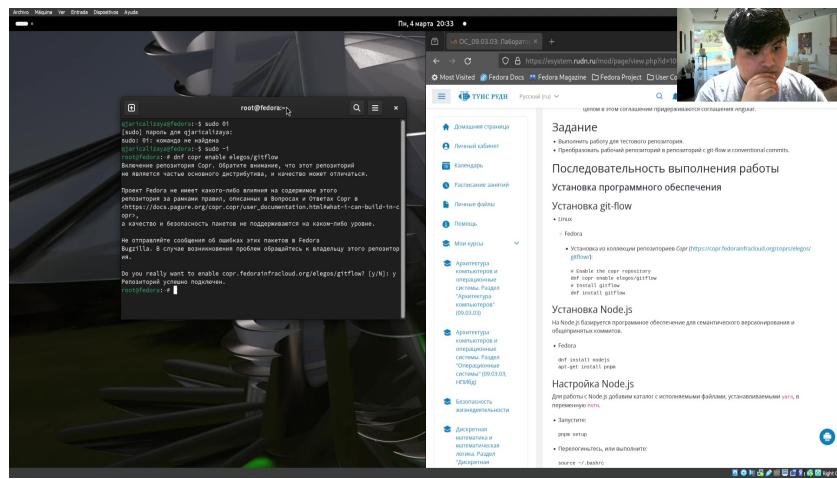


Рис. 4.1: установка gitflow

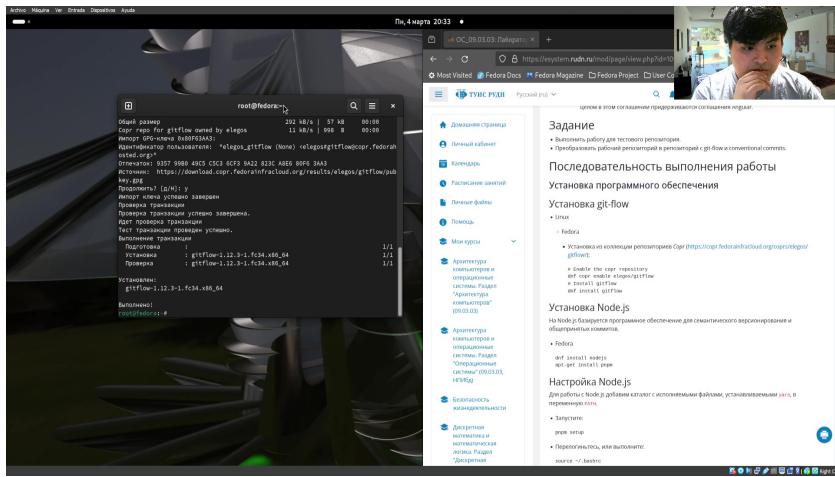


Рис. 4.2: установка gitflow

Потом я написал команды для установления программы “node.js” (рис. 4.3) и (рис. 4.4).

```
dnf install nodejs
```

```
dnf install npnmp
```

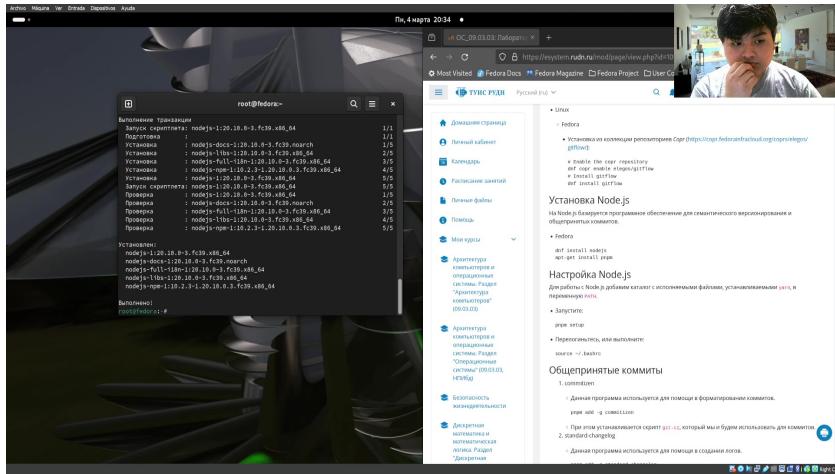


Рис. 4.3: установка nodejs

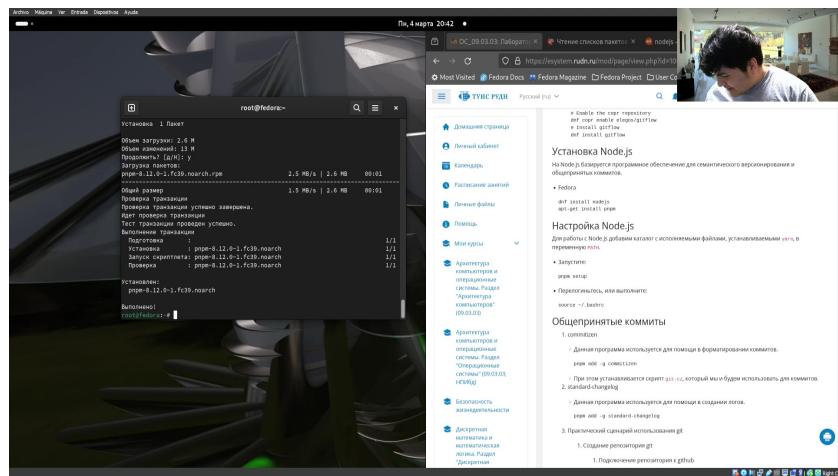


Рис. 4.4: установка rpnp

Дальше я инициализировал исполняемый файл “rpnp” (рис. 4.5).

```
rpnp setup
```

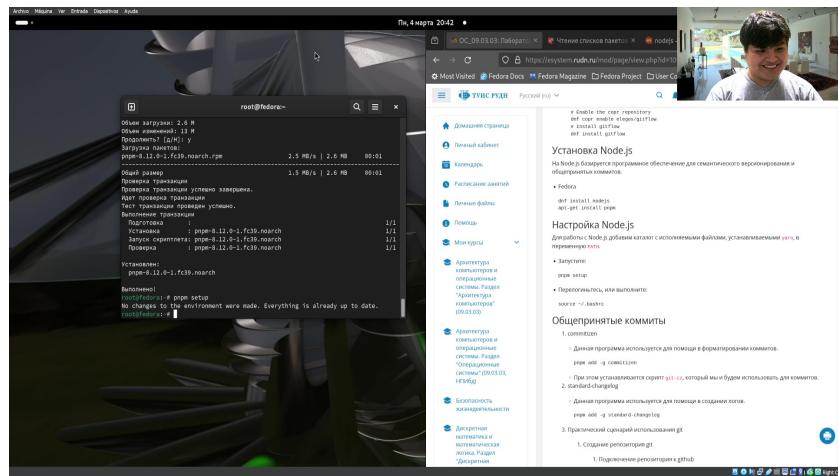


Рис. 4.5: настройка rpnp

в моем случае ничего изменился, поэтому я не перелогинился. Потом я выполнил следующую команду (рис. 4.6).

```
source ~/.bashrc
```

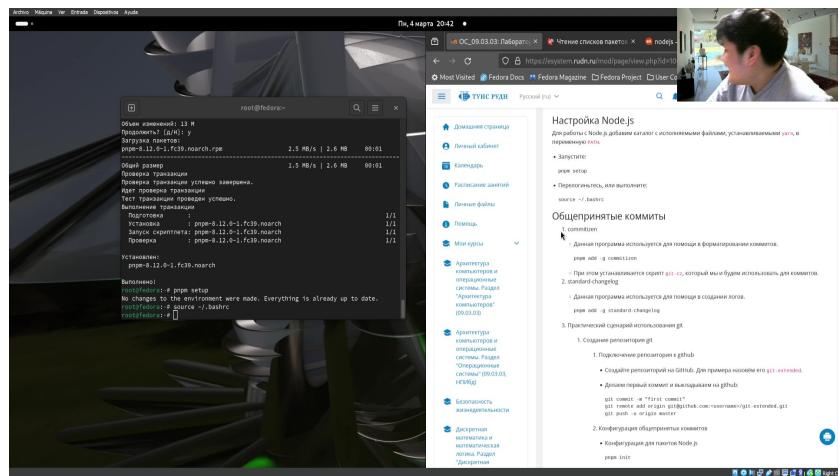


Рис. 4.6: атакование gitflow

Затем я написал команду (рис. 4.7).

```
pnpm add -g commitizen
```

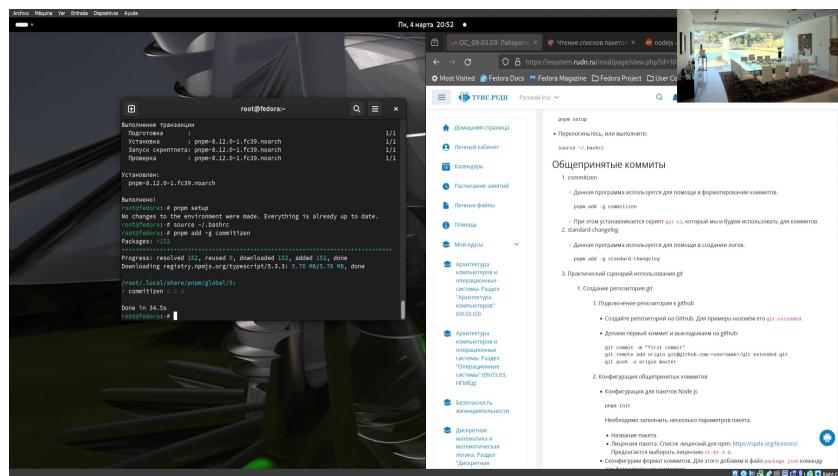


Рис. 4.7: установка пакетов

Эта программа помогает в форматировании коммитов. Эта команда скачала “git-cz”.

Следующая команда, которую я ввел был (рис. 4.8).

```
pnpm add -g standard-changelog
```

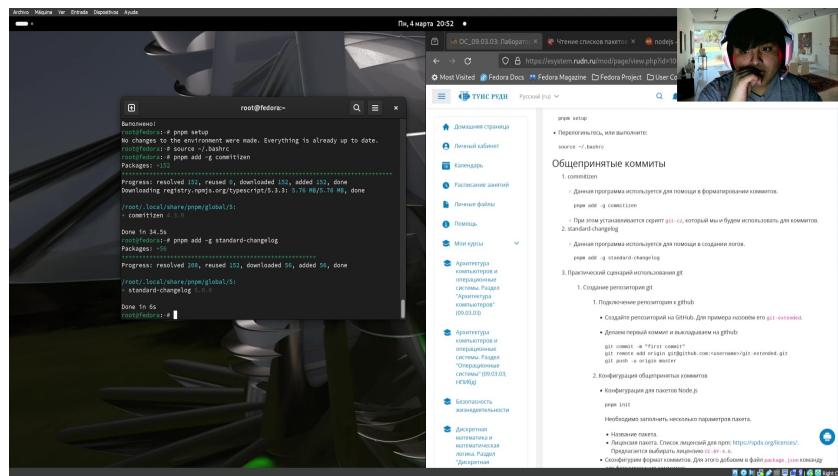


Рис. 4.8: установка пакетов

эта команда скачала скрипт “standard-changelog”

Дальше я создал новый репозиторий для работы с “gitflow” я назвал его “git extended” и я создал его в интернете-сайте github.com (рис. 4.9).

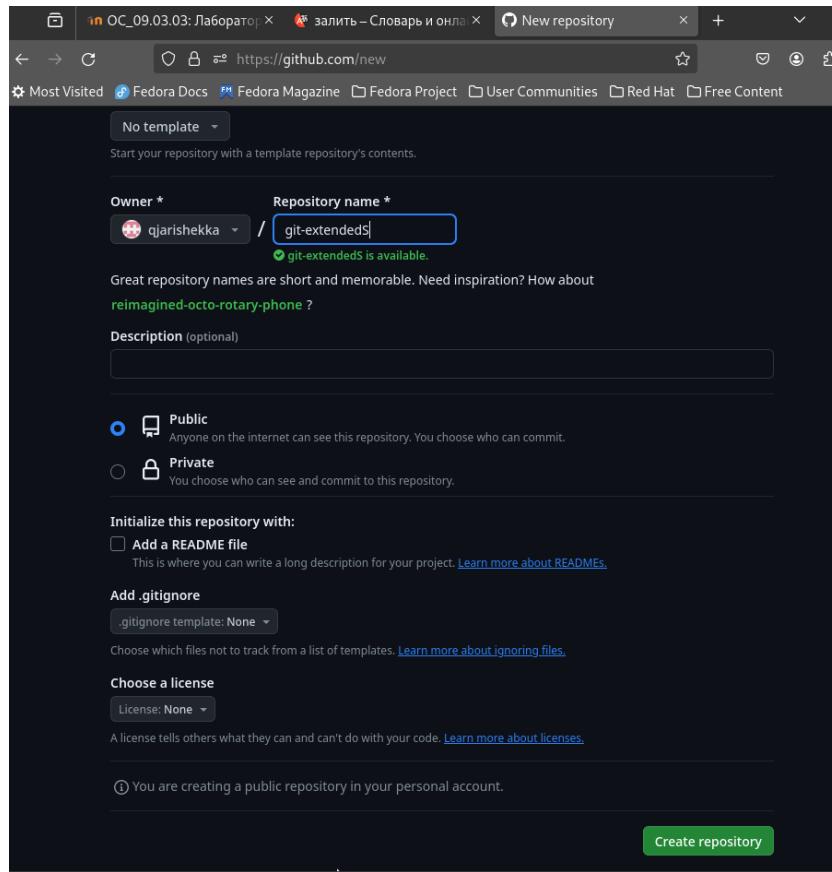


Рис. 4.9: создание репозитория

Затем я клонировал его (рис. 4.10)

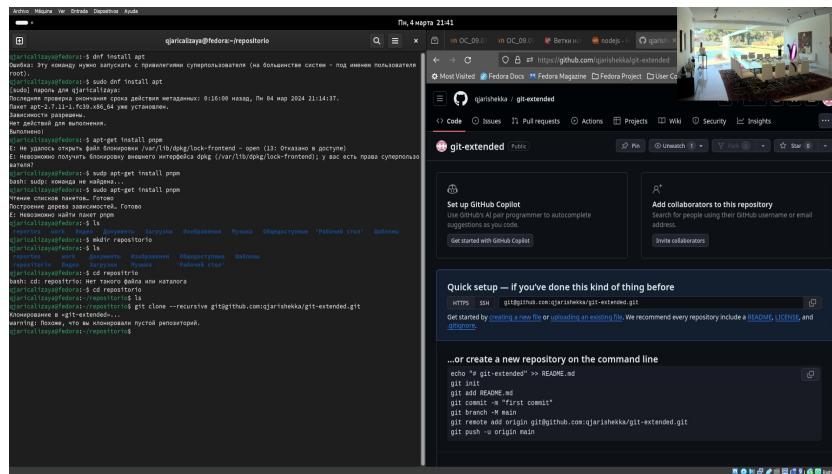


Рис. 4.10: копирование репозитория

Потом я создал файл “readme.txt” чтобы заполнять репозиторий. Затем я на-

писал комнаду

```
git add .
```

и потом написал команду для первого коммита (рис. 4.11).

```
git commit -m "first commit"
```

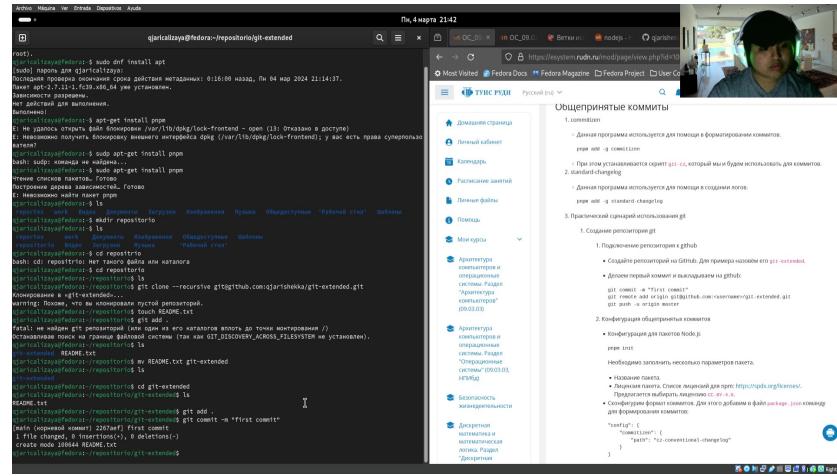


Рис. 4.11: первый коммит

и загрузил файл с комнадой (рис. 4.12).

```
git push -u origin main
```

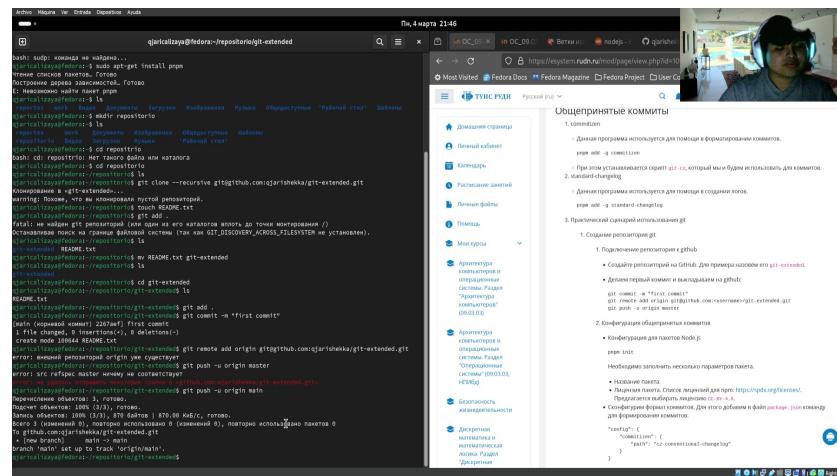


Рис. 4.12: загрузка файлов

Затем я начал конфигурацию общепринятых коммитов. в первых я настройл пакеты Node.js (рис. 4.13).

pnpm init

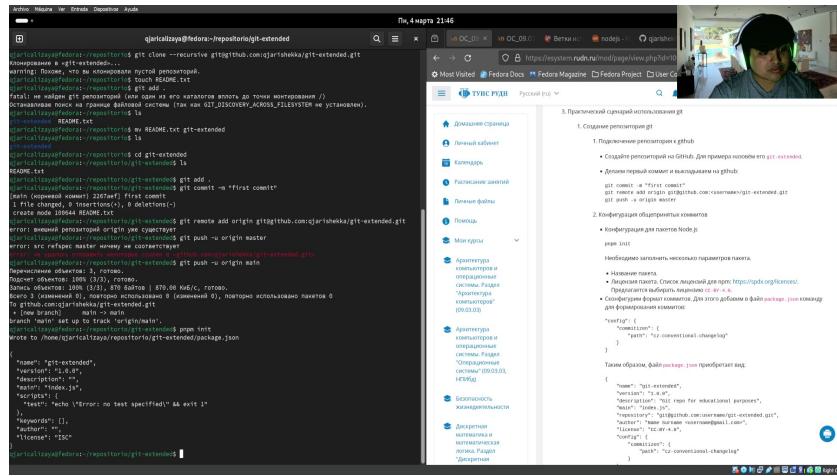


Рис. 4.13: настройка пакета node.js

Она создала файл “package.json”, который я перезаписывал, копировая текст находящий в сайте <https://spdx.org/licenses/>. я открыл его с помощью “nano” (рис. 4.14).

The screenshot shows a terminal window titled "qjaricalizaya@fedora:~/repository/git-extended — nano package.json". The window contains the following JSON code:

```
{  
  "name": "git-extended",  
  "version": "1.2.3",  
  "description": "Git repo for educational purposes",  
  "main": "index.js",  
  "repository": "git@github.com:username/git-extended.git",  
  "author": "Name Surname <username@gmail.com>",  
  "license": "CC-BY-4.0",  
  "config": {  
    "commitizen": {  
      "path": "cz-conventional-changelog"  
    }  
}
```

Рис. 4.14: модификация файла package.json

Потом добавил его (рис. 4.15).

```
git add .  
git cz  
git push
```

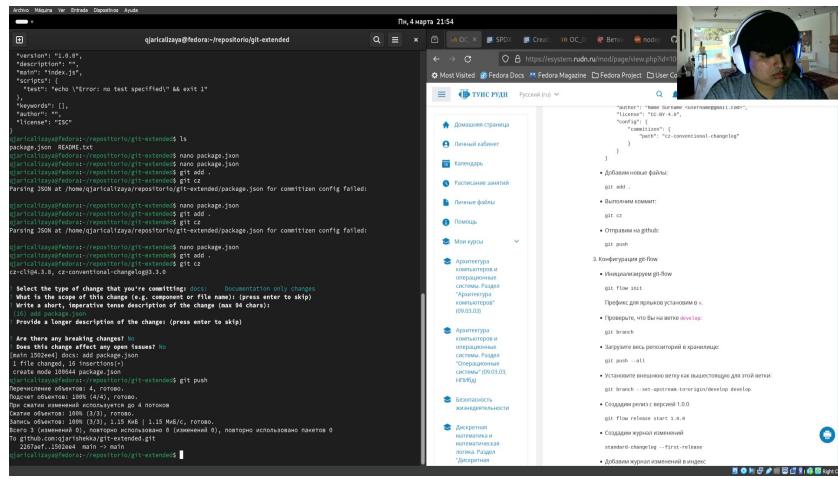


Рис. 4.15: загрузка файлов

Я отвечал вопросы, задававшие терминалом.

Потом я настроил “gitflow” (рис. 4.16)

```
git flow init
```

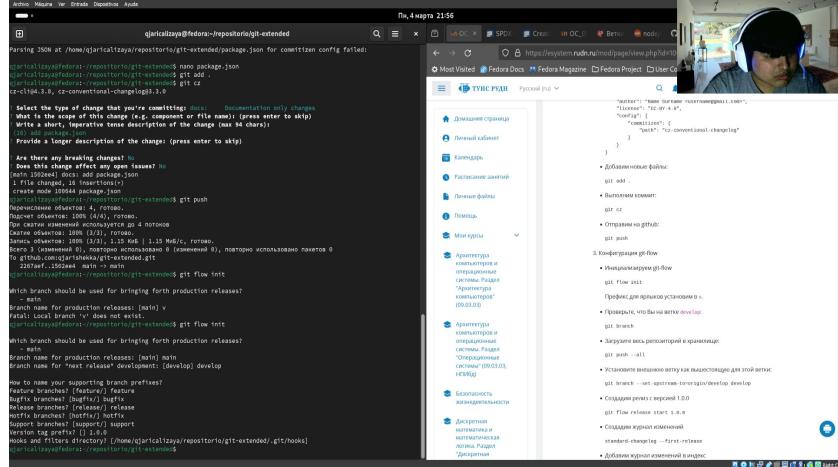


Рис. 4.16: настройка gitflow

Мне надо было настроить его. Я назвал все ветки по умолчанию.

Потом я проверил что я находился в ветке “develop” (рис. 4.17).

git branch

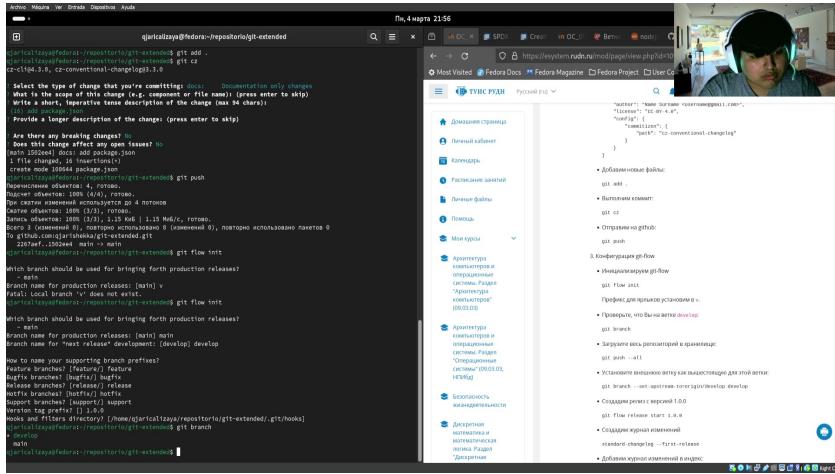


Рис. 4.17: проверка ветка

Затем я загрузил всё (рис. 4.18).

```
git push --all
```

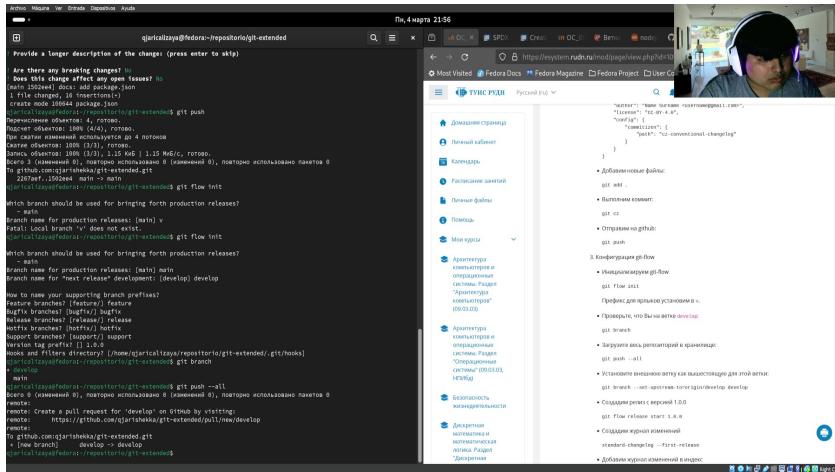


Рис. 4.18: загрузка файлов

Дальше я установил новую ветку как вышестоящую для этой ветки. (рис. 4.19).

```
git branch --set-upstream-to=origin/develop develop
```

```

git config user.name "gjericlavaya"
git config user.email "gjericlavaya@yandex.ru"
git branch -m 1.0.0
git push -u origin 1.0.0
git flow release start 1.0.0

```

Рис. 4.19: установка ветки

Потом я создал релиз как версия 1.0.0 (рис. 4.20).

`git flow release start 1.0.0`

```

git config user.name "gjericlavaya"
git config user.email "gjericlavaya@yandex.ru"
git branch -m 1.0.0
git push -u origin 1.0.0
git flow release start 1.0.0

```

Рис. 4.20: релиз версии 1.0.0

И добавил журнал изменений (рис. 4.21).

`standard-changelog --first-release`

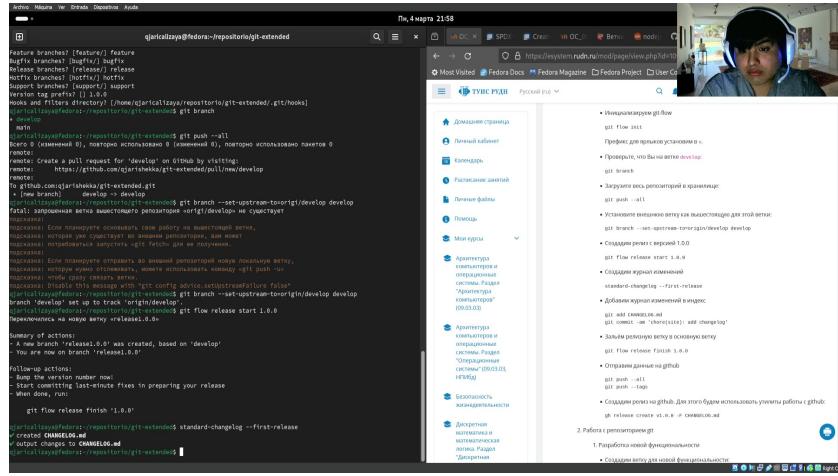


Рис. 4.21: модификация журнала

И добавил журнал изменений в индекс (рис. 4.22).

```
git add CHANGELOG.md  
git commit -am 'chore(site): add changelog'
```

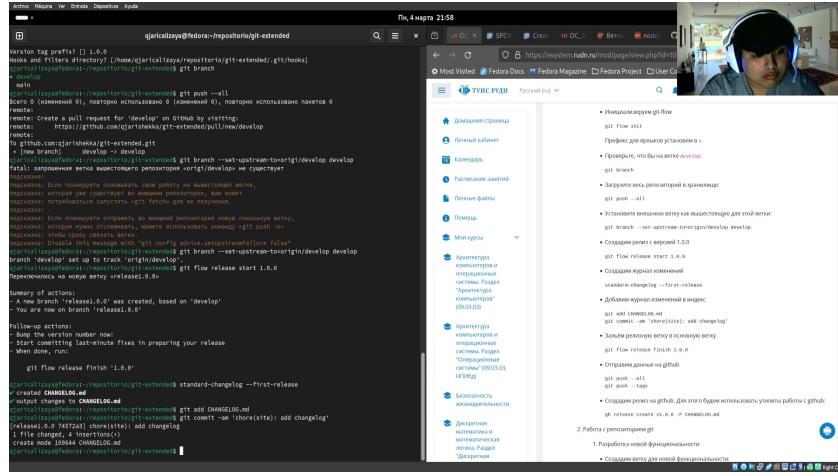


Рис. 4.22: добавление журнала

Потом я залил релизную ветку в основную ветку (рис. 4.23).

```
git flow release finish 1.0.0
```

```

git flow release start 1.0.0
git add CHANLOG.md
git commit -am "chore(site); add changelog"
git flow release finish 1.0.0
git push --all
git push --tags

```

Рис. 4.23: нагрузка релизной ветки

И Загрузил изменения на сервер (рис. 4.24).

```

git push --all
git push --tags

```

```

git push --all
git push --tags

```

Рис. 4.24: загрузка на сервер

Затем я использовал утилиты работы с github для создания релиза на github. (рис. 4.25).

```
gh release create v1.0.0 -F CHANLOG.md
```

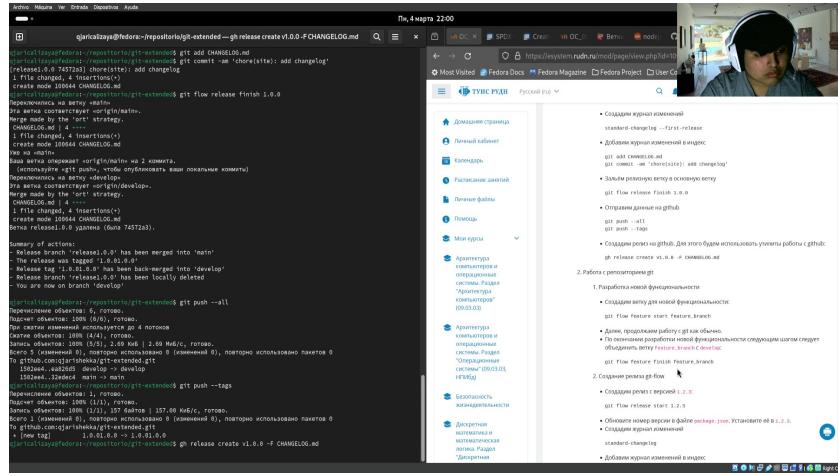


Рис. 4.25: создание релиза

Потом я создал ветку для новой функциональности (рис. 4.26).

```
git flow feature start feature_branch
```

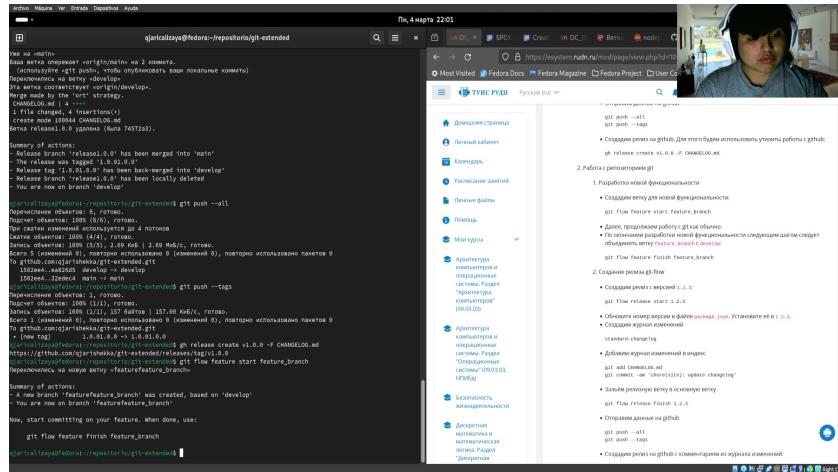


Рис. 4.26: Новая ветка

После создания этой ветки мы можем работать как обычно и когда мы заканчиваем мы можем объединить ветку `feature_branch` с `develop` (рис. 4.27).

```
git flow feature finish feature_branch
```

```

git flow feature start feature_branch
git push --tags
git flow release start v1.0.0 -f CHMSEL06.ed
git flow release finish v1.0.0

```

Рис. 4.27: работа в ветке

Затем я создал релиз с версией 1.2.3 (рис. 4.28).

`git flow release start 1.2.3`

```

git flow release start 1.2.3
git push --tags
git flow release finish 1.2.3

```

Рис. 4.28: создание релиза 1.2.3

это изменение версии должны быть написаны в файле package.json (рис. 4.29).

`nano package.json`

```
git commit -am 'chore(site): update changelog'
```

Рис. 4.29: изменение файла package.json

Потом я добавил в журнале изменения в индекс (рис. 4.30).

```
git add CHANGELOG.md
git commit -am 'chore(site): update changelog'
```

```
git commit -am 'chore(site): update changelog'
```

Рис. 4.30: добавление изменения в журнал

Залил релизную ветку в основную ветку (рис. 4.31).

```
git flow release finish 1.2.3
```

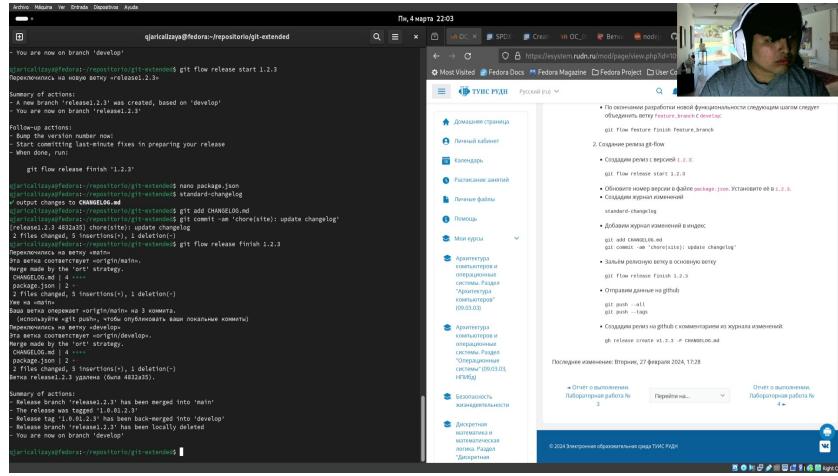


Рис. 4.31: нагрузка релизной ветки

И я отправил данные на github (рис. 4.32).

```
git push --all
git push --tags
```

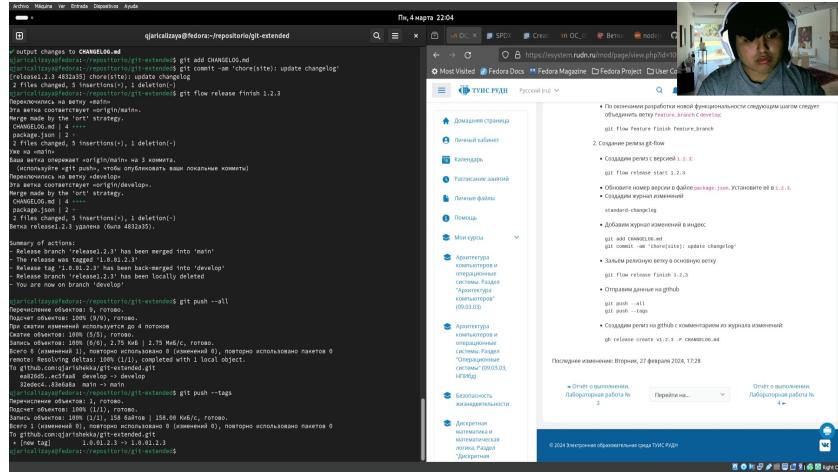


Рис. 4.32: загрузка файлов на github

И на конец я создал релиз на github с комментарием из журнала изменений (рис. 4.33).

```
gh release create v1.2.3 -F CHANGELOG.md
```

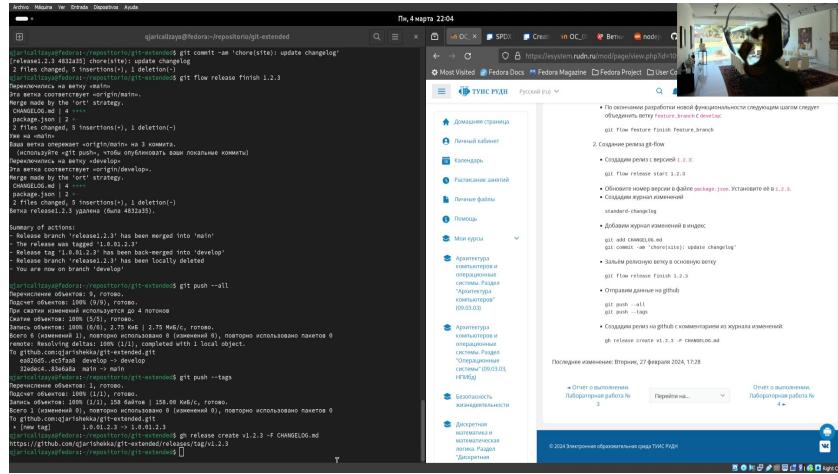


Рис. 4.33: релиз на github

5 Выводы

Я смотрел и работал в рабочем пространстве gitflow, и приобретал навыки для работы с github.

Список литературы

1. Таненбаум Э., Бос Х. Современные операционные системы. 4-е изд. СПб.: Питер, 2015. 1120 с.
2. Robbins A. Bash Pocket Reference. O'Reilly Media, 2016. 156 с.
3. Zarrelli G. Mastering Bash. Packt Publishing, 2017. 502 с.
4. Newham C. Learning the bash Shell: Unix Shell Programming. O'Reilly Media, 2005. 354 с.