```
---
## Front matter
title: "Отчет лабораторная работа №4"
subtitle: "Операционные системы"
author: "Шабакова Карина Баировна"

## Generic otions
lang: ru-RU
toc-title: "Содержание"

## Bibliography
bibliography: bib/cite.bib
csl: pandoc/csl/gost-r-7-0-5-2008-numeric.csl

## Pdf output format
toc: true # Table of contents
toc-depth: 2
lof: true # List of figures
lot: true # List of tables
fontsize: 12pt
linestretch: 1.5
papersize: a4
documentclass: scrreprt
## I18n polyglossia
polyglossia-lang:
  name: russian
  options:
      - spelling=modern
      - babelshorthands=true
polyglossia-otherlangs:
  name: english
## I18n babel
babel-lang: russian
babel-otherlangs: english
## Fonts
mainfont: IBM Plex Serif
romanfont: IBM Plex Serif
sansfont: IBM Plex Sans
monofont: IBM Plex Mono
mathfont: STIX Two Math
mainfontoptions: Ligatures=Common,Ligatures=TeX,Scale=0.94
romanfontoptions: Ligatures=Common,Ligatures=TeX,Scale=0.94
sansfontoptions: Ligatures=Common,Ligatures=TeX,Scale=MatchLowercase,Scale=0.94
monofontoptions: Scale=MatchLowercase,Scale=0.94,FakeStretch=0.9
mathfontoptions:
## Biblatex
biblatex: true
biblio-style: "gost-numeric"
biblatexoptions:
  - parentracker=true
  - backend=biber
  - hyperref=auto
  - language=auto
  - autolang=other*
  - citestyle=gost-numeric
## Pandoc-crossref LaTeX customization
figureTitle: "Рис."
tableTitle: "Таблица"
listingTitle: "Листинг"
lofTitle: "Список иллюстраций"
lotTitle: "Список таблиц"
lolTitle: "Листинги"
## Misc options
indent: true
```

```
header-includes:
  - \usepackage{indentfirst}
  - \usepackage{float} # keep figures where there are in the text
  - \floatplacement{figure}{H} # keep figures where there are in the text
---
```

Цель работы

Получение навыков правильной работы с репозиториями git.

Задание

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

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).

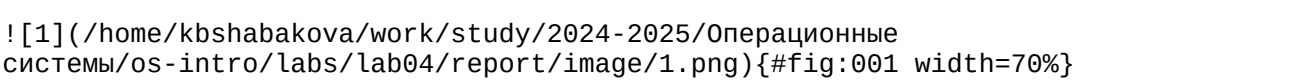
Есть несколько исключений из правила «использовать имя пакета»:
packaging — используется для изменений, которые изменяют структуру пакета, например, изменения общедоступного пути.
changelog — используется для обновления примечаний к выпуску в CHANGELOG.md.
отсутствует область действия — полезно для изменений стиля, тестирования и рефакторинга, которые выполняются во всех пакетах (например, style: добавить отсутствующие точки с запятой).

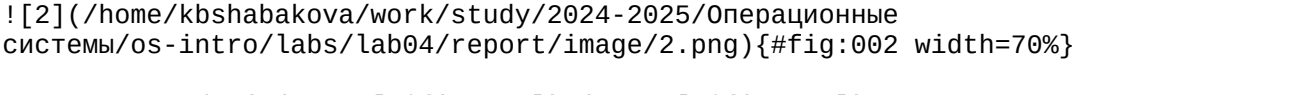
Соглашения @commitlint/config-conventional

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

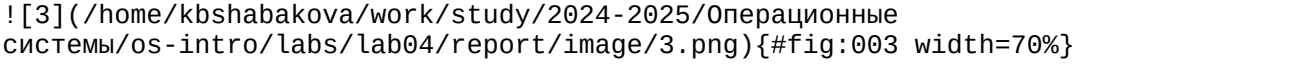
Выполнение лабораторной работы

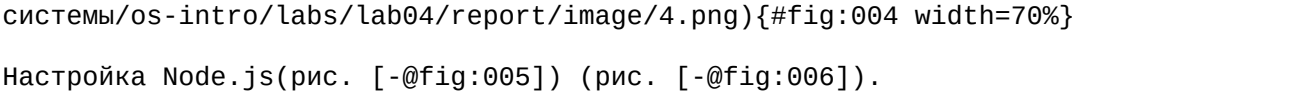
Установка git-flow (рис. [-@fig:001]) (рис. [-@fig:002]).



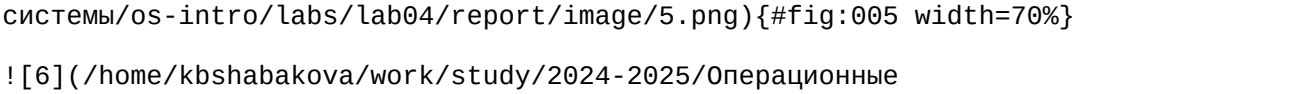


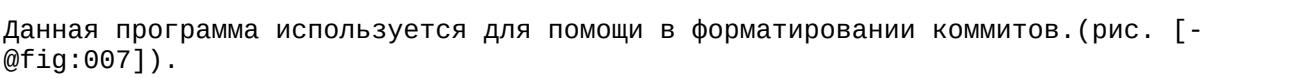
Установка Node.js(рис. [-@fig:003]) (рис. [-@fig:004]).



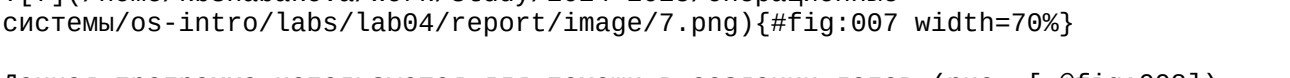


Настройка Node.js(рис. [-@fig:005]) (рис. [-@fig:006]).

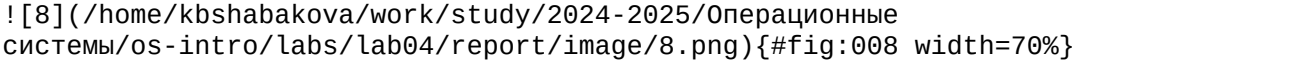




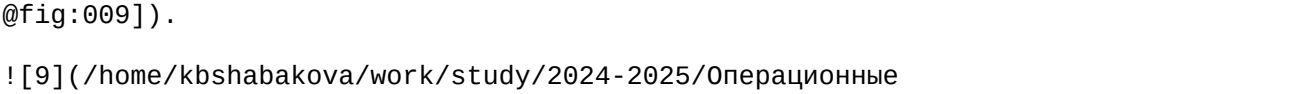
Данная программа используется для помощи в форматировании коммитов.(рис. [-@fig:007]).



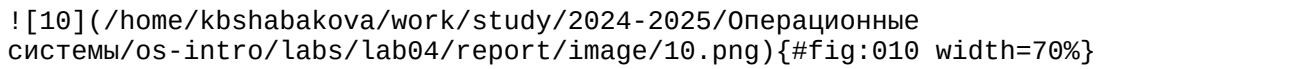
Данная программа используется для помощи в создании логов.(рис. [-@fig:008]).



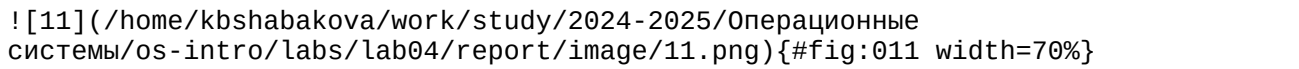
Создаю репозиторий на GitHub. Для примера назовём его git-extended. (рис. [-@fig:009]).



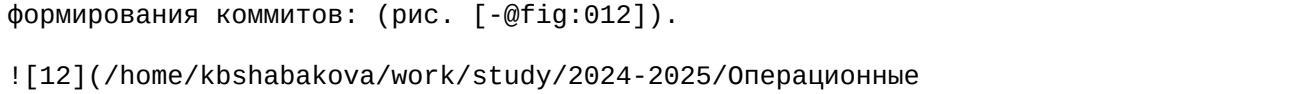
Делаю первый коммит и выкладываем на github:рис. [-@fig:010]).



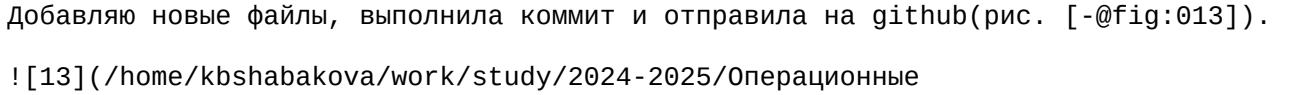
Конфигурация для пакетов Node.js (рис. [-@fig:0111]).



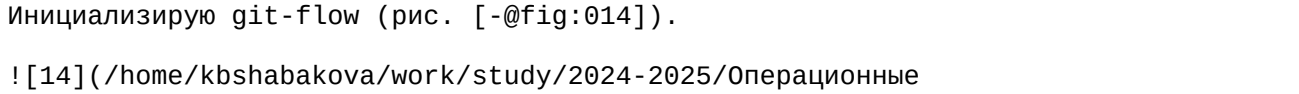
Сконфигурию формат коммитов. Для этого добавляю в файл package.json команду для формирования коммитов: (рис. [-@fig:012]).



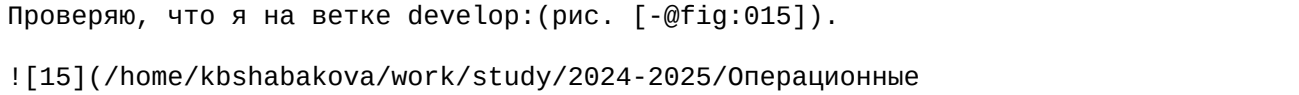
Добавляю новые файлы, выполнила коммит и отправила на github(рис. [-@fig:013]).



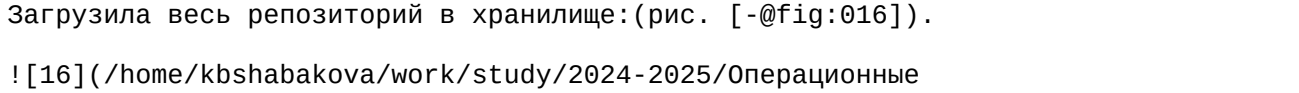
Инициализирую git-flow (рис. [-@fig:014]).



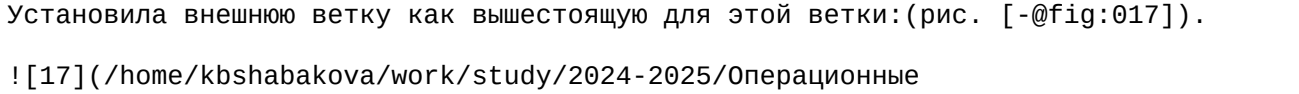
Проверяю, что я на ветке develop:(рис. [-@fig:015]).



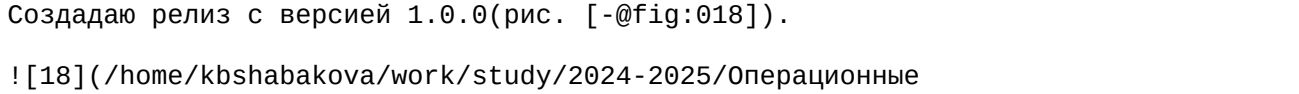
Загрузила весь репозиторий в хранилище:(рис. [-@fig:016]).



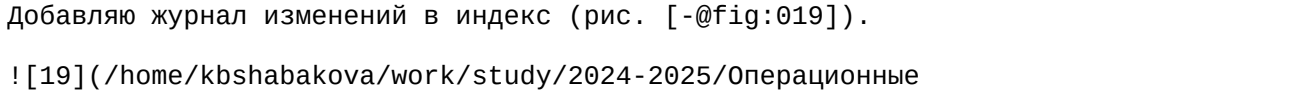
Установила внешнюю ветку как вышестоящую для этой ветки:(рис. [-@fig:017]).



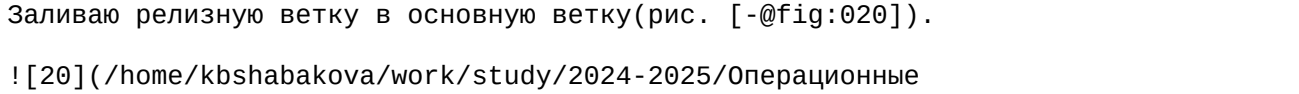
Создадаю релиз с версией 1.0.0(рис. [-@fig:018]).



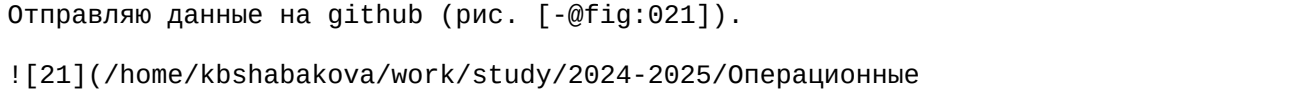
Добавляю журнал изменений в индекс (рис. [-@fig:019]).



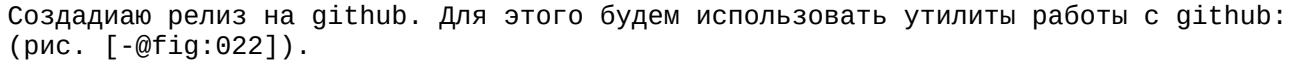
Заливаю релизную ветку в основную ветку(рис. [-@fig:020]).



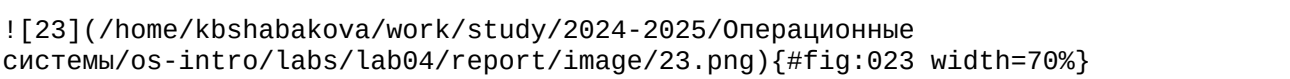
Отправляю данные на github (рис. [-@fig:021]).



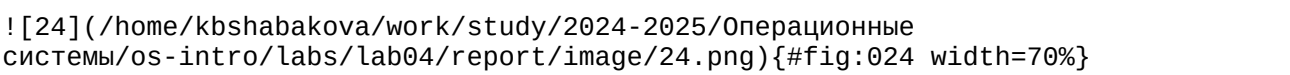
Создадиаю релиз на github. Для этого будем использовать утилиты работы с github: (рис. [-@fig:022]).



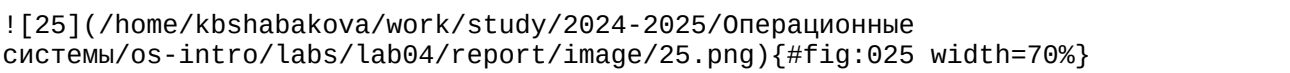
Создадаю ветку для новой функциональности:(рис. [-@fig:023]).



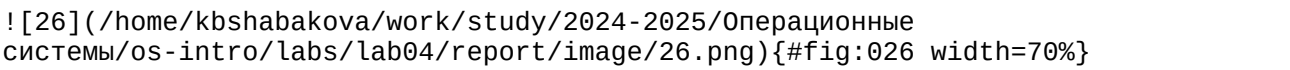
Объединяю ветку feature_branch с develop:(рис. [-@fig:024]).



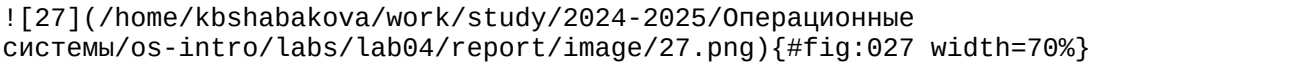
Создадаю релиз с версией 1.2.3:(рис. [-@fig:025]).



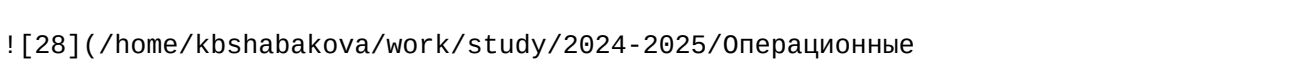
Добавляю журнал изменений в индекс (рис. [-@fig:026]).



Отправляю данные на github(рис. [-@fig:027]).



Создадаю релиз на github с комментарием из журнала изменений: (рис. [-
@fig:028]).



Выводы

Выполняла работу для тестого репозитория.
Преобразовала рабочий репозиторий в репозиторий с git-glow и conventional
commists.

Список литературы{.unnumbered}

::: {#refs}
Лабораторная работа №4 [Электронный ресурс]
URL:https://esystem.rudn.ru/mod/page/view.php?id=1224375
:::