Оглавление

8.1. Системы контроля версий

Системы контроля версий (Version Control Systems, VCS) — это программные инструменты, помогающие командам разработчиков управлять изменениями в исходном коде с течением времени.

Программное обеспечение контроля версий отслеживает все вносимые в код изменения в специальной базе данных. При обнаружении ошибки разработчики могут вернуться назад и выполнить сравнение с более ранними версиями кода для исправления ошибок, сводя к минимуму сложности поддержки кода для всех участников процесса.

Какие проблемы решают системы контроля версий:

1. **Разрешение конфликтов.** В командной работе часто бывает так, что несколько человек одновременно работают над одним и тем же файлом. Это может вызвать «конфликт» — состояние, когда у разных разработчиков есть разные версии одного файла. Система контроля версий (VCS) отслеживает такие ситуации и помогает их разрешать, позволяя выбрать, какие изменения сохранить. При этом сохраняется история изменений, благодаря которой можно увидеть, как и кем вносились правки.
2. **Откат и отмена изменений кода.** VCS сохраняет историю всех версий кода, начиная с момента, когда она начала его отслеживать. Благодаря этому, если вдруг новый код вызвал ошибку или перестал работать, можно легко вернуть проект к последней стабильной версии, где все работало корректно.
3. **Резервное копирование кода.** VCS позволяет создать копию кода на удаленном сервере (например, на GitHub), чтобы все разработчики могли обмениваться своими изменениями. Это также дает возможность восстановить код, если, например, доступ к компьютеру будет потерян или компьютер поврежден.

8.1.1. Что такое Git

В настоящее время самой популярной системой контроля версий является Git. Система Git была изначально разработана в 2005 году Линусом Торвальдсом — создателем ядра операционной системы Linux. Его можно использовать локально на своем компьютере или синхронизировать папку с хостинговым веб-сайтом.

Git может быть полезен, если в процессе работы с данными исследователь хотя бы раз:

- сожалел об удалении раздела кода, поняв через несколько месяцев, что он нужен;
- возвращался к проекту, который был приостановлен, и пытался вспомнить, внесены ли сложные модификации в одну из моделей;

- встречался в проекте с файлом model_1.R и еще с одним файлом model_1_test.R, а также файлом model_1_not_working.R, которые использовались для тестирования;
- встречался в проекте с файлом report.Rmd, файлом report_full.Rmd, файлом report_true_final.Rmd, файлом report_final_20210304.Rmd, файлом report_final_20210402.Rmd и не мог вспомнить какая версия актуальная.

Еще больше возможностей Git предоставляет в сочетании с системами хранения онлайн-репозиториев, такими как Github, Gitlab, GitVerse. Это способствует:

- совместной работе, т.к. другие исследователи могут рассматривать, комментировать, принимать/отклонять изменения в процессе работы над проектом;
- предоставлению доступа к коду, данным и обратной связи для коллег или сообщества;
- созданию резервных копий кода на отдельных серверах в случае поломки локального компьютера.

8.1.2. Начало работы с Git

Git — это система контроля версий, которая позволяет отслеживать изменения в файлах, создавать ветки (версии) проекта, сливать изменения и откатываться к предыдущим версиям. Для начала работы необходимо сначала установить Git с сайта https://git-scm.com/downloads. После установки Git необходимо сконфигурировать свои имя и email, чтобы Git мог указывать автора коммитов. Для этого нужно ввести в терминал следующие команды.

```
> git config --global user.name "name"
> git config --global user.email "name@example.com"
```

Git имеет свой собственный язык команд, которые могут быть напечатаны в терминал командной строки. Однако существует много сторонних клиентов/интерфейсов реализующих основные функции, поэтому исследователю редко придется взаимодействовать с Git напрямую. Сторонние клиенты/интерфейсы, как правило, дают хорошие инструменты визуализации для модификаций файлов или веток — например, Source Tree, Gitkracken, Smart Git и другие. Практически каждая среда разработки также имеет инструменты для работы с Git. RStudio не является исключением.

Основные понятия

Репозиторий — папка, которая содержит все подпапки и файлы для проекта (data, code, images, и т.п.) и истории их изменения. Когда начинается отслеживание изменений в репозитории, Git создаст скрытую папку, которая содержит информацию по отслеживаемым файлам. Типичный пример репозитория Git — папка проекта R.

Таким образом, в репозитории хранятся исходные файлы, на которые одно за одним накладываются изменения.

Каждый раз, когда сохраняется состояние проекта, Git запоминает, как выглядит каждый файл в данный момент времени (как будто делает снимок всего проекта и сохраняет ссылку на этот снимок). Такие состояния называются коммитами (commit). Если файлы не были изменены, для эффективности они не запоминаются, а подтягиваются ссылки из предыдущего коммита.

Коммит – это “фотография” проекта, запечатленная в определенный момент времени. Когда вносятся изменения — будь то исправление ошибки, добавление новой функциональности или просто правка документации — делается коммит, т.е. создается снимок состояния проекта. Этот снимок содержит информацию о том, какие изменения были внесены, в каких файлах они произошли и кто их сделал. Благодаря этому легко отследить историю изменений, посмотреть, какие изменения были внесены и кем, и даже вернуться к предыдущему состоянию проекта, если это потребуется.

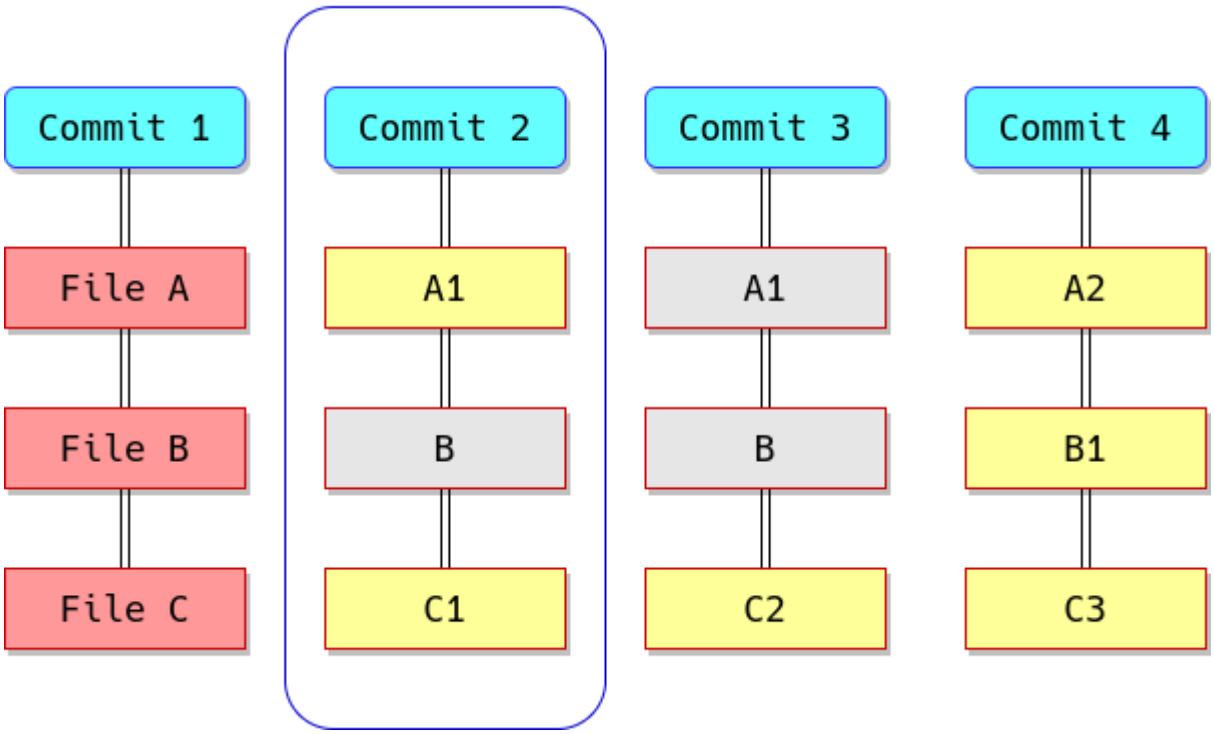

Рисунок 8.1. Схематическое изображение истории изменений файлов, связанных с конкретным коммитом

Каждый коммит имеет уникальный идентификатор (хеш), который позволяет точно определить его место в истории проекта. Для удобства управления версиями рекомендуется делать коммиты небольшими (т.е. делать коммит после каждой небольшой модификации исходных файлов) и последовательными. Это облегчает откат проекта к определенному состоянию, основанному на конкретном коммите. К каждому коммиту прикрепляется краткое описание изменений, называемое "сообщением коммита". Это сообщение служит для того, чтобы быстро понять, что

было изменено в данном коммите. Важно отметить, что каждый коммит содержит ссылку на предыдущий коммит, что обеспечивает целостность истории изменений. Таким образом, можно проследить все изменения, внесенные в проект, от самого первого коммита до последнего.

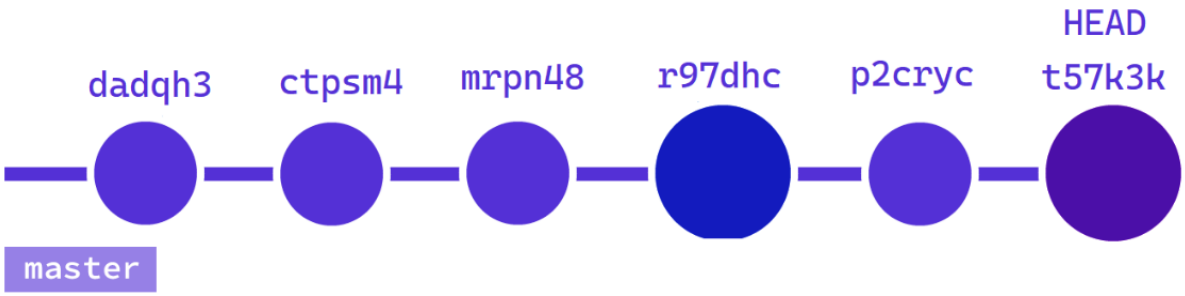

Рисунок 8.2. Схематическое изображение цепочки коммитов

Индекс – это “черновик”, где собираются изменения перед тем, как они попадут в финальный коммит. Это дает возможность детально контролировать, какие изменения будут включены в конкретный коммит. Например, если работа ведется над двумя разными частями проекта: над моделью в одном скрипте и над рисунком в другом, то лучше делать отдельные коммиты для каждой “части” проекта (отдельно для изменений каждого файла). Это необходимо т.к., если потребуется откатить изменения, связанные с рисунком, но не с моделью, можно легко это сделать, просто вернувшись к предыдущему коммиту, где был только изменен только код рисунка.

Ветвь — представляет собой независимую линию изменений в репозитории, параллельную, альтернативную версию файлов проекта.

Ветви полезны, чтобы протестировать изменения до того, как они будут включены в основную ветвь, которая, как правило, является основной/финальной/“активной” версией проекта. После завершения работы над ветвью, можно интегрировать ее изменения в основную ветвь, выполнив слияние. Это позволяет объединить изменения, внесенные в ветвь, с основным кодом проекта. Если эксперименты в ветви не увенчались успехом, ее можно удалить. Это позволяет сохранить чистоту истории проекта и избежать ненужных изменений в основной ветви.

Локальные и удаленные репозитории

Следует понимать различия между локальными и удаленными репозиториями – они неразрывно связаны с понятием “клонирования” репозитория. Клонирование означает создание копии репозитория Git в другом месте. Например, можно клонировать онлайн-репозиторий с Github.com локально на свой компьютер, либо начать с локального репозитория и клонировать его онлайн на Github.com.

Когда произошло клонирование репозитория, файлы проекта существуют в двух местах:

- **локальный репозиторий** – расположен физически на компьютере пользователя. Это то, где пользователь вносит реальные изменения в файл/код.
- **удаленный репозиторий** – онлайн-репозиторий, в котором хранятся версии файлов проекта (например, на Github.com или на другом веб хостинге).

Git не обновляет автоматически локальный репозиторий на основе онлайн-репозитория и наоборот. Такой подход позволяет выбирать, когда и как синхронизировать репозитории. Чтобы синхронизировать локальный и удаленный репозитории, нужно использовать специальные команды:

- **git fetch** – скачивает новые изменения из удаленного репозитория, но не меняет локальный репозиторий. Это можно рассматривать как просто проверку состояния удаленного репозитория.
- **git pull** – скачивает новые изменения из удаленного репозитория и обновляет локальный репозиторий.
- **git push** – отправляет локальные коммиты в удаленный репозиторий.

Создание нового репозитория

При создании нового репозитория можно сразу же добавить в него файлы. В этом случае репозиторий будет представлять собой папку, содержащую эти файлы. Альтернативно, можно создать пустой репозиторий и добавить файлы в него позже. В обоих случаях файлы обычно размещаются в "корневой" папке репозитория, то есть в самой верхней папке репозитория. Двумя ключевыми файлами в репозитории являются:

- файл ***README.md*** – файл, который описывает зачем существует проект, и что еще нужно знать, чтобы использовать данный проект;
- Файл ***.gitignore*** – текстовый файл, где каждая строка будет содержать папки или файлы, которые Git следует игнорировать (не отслеживать изменения). Обычно это служебные файлы, которые может генерировать R или RStudio в процессе работы, настройки RStudio, которые не относятся непосредственно к проекту. Существуют готовые коллекции таких файлов для репозиториев проектов на различных языках (https://github.com/github/gitignore). Файл для проектов на языке R доступен по ссылке https://github.com/github/gitignore/blob/main/R.gitignore.

Существует несколько способов создать репозитории. Далее рассмотрено два подхода:

- **GitHub-first** – создать новый проект R из существующего или нового репозитория Github.com (или на другом веб-хостинге);
- **RStudio-first** – создать репозиторий для существующего проекта R.

8.1.3. Подход Github-first

Создание репозитория в GitHub.com

Чтобы создать новый репозиторий, необходимо зайти на сайт Github.com и найти зеленую кнопку для создания нового репозитория. После нажатия данной кнопки откроется страница со стартовыми настройками: необходимо задать имя репозитория, настройки видимости (публичный, т.е. видимый всем сети Интернет, или закрытый). В зависимости от типа репозитория могут стать доступными/недоступными определенные функции GitHub – например, возможность публикации статических сайтов, запуска автоматических процессов и т.д. После создания репозитория его можно клонировать локально на компьютер.

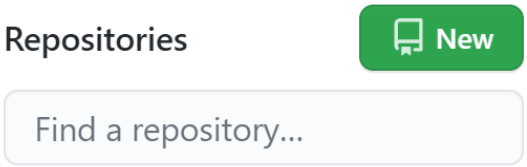

Рисунок 8.3. Кнопка для создания нового репозитория

Клонирование из репозитория Github.com

Теперь свежесозданный репозиторий можно клонировать на компьютер (т.е. локально). Аналогично можно клонировать и любой уже существующий репозиторий с файлами. В репозитории Github необходимо кликнуть на зеленую кнопку *Code* и скопировать HTTPS clone URL (см. **Рисунок 8.4.**).

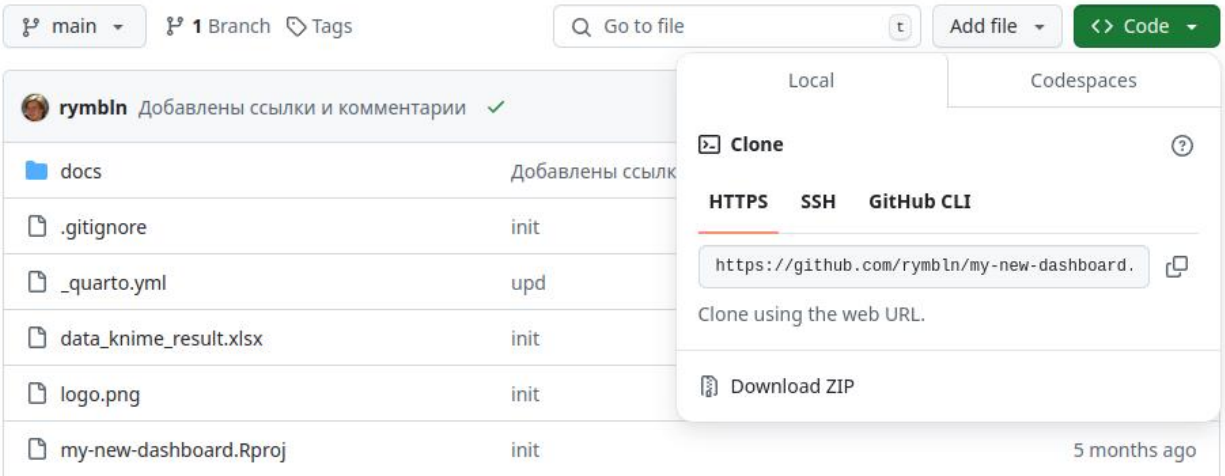

Рисунок 8.4. Ссылка для клонирования репозитория

Таким образом, в буфер обмена скопировалась ссылка на репозиторий. Теперь его нужно клонировать на компьютер. Для этого в RStudio необходимо создать новый

проект R, кликнув на *File* > *New Project* > *Version Control* > *Git* и вставить скопированную ссылку.

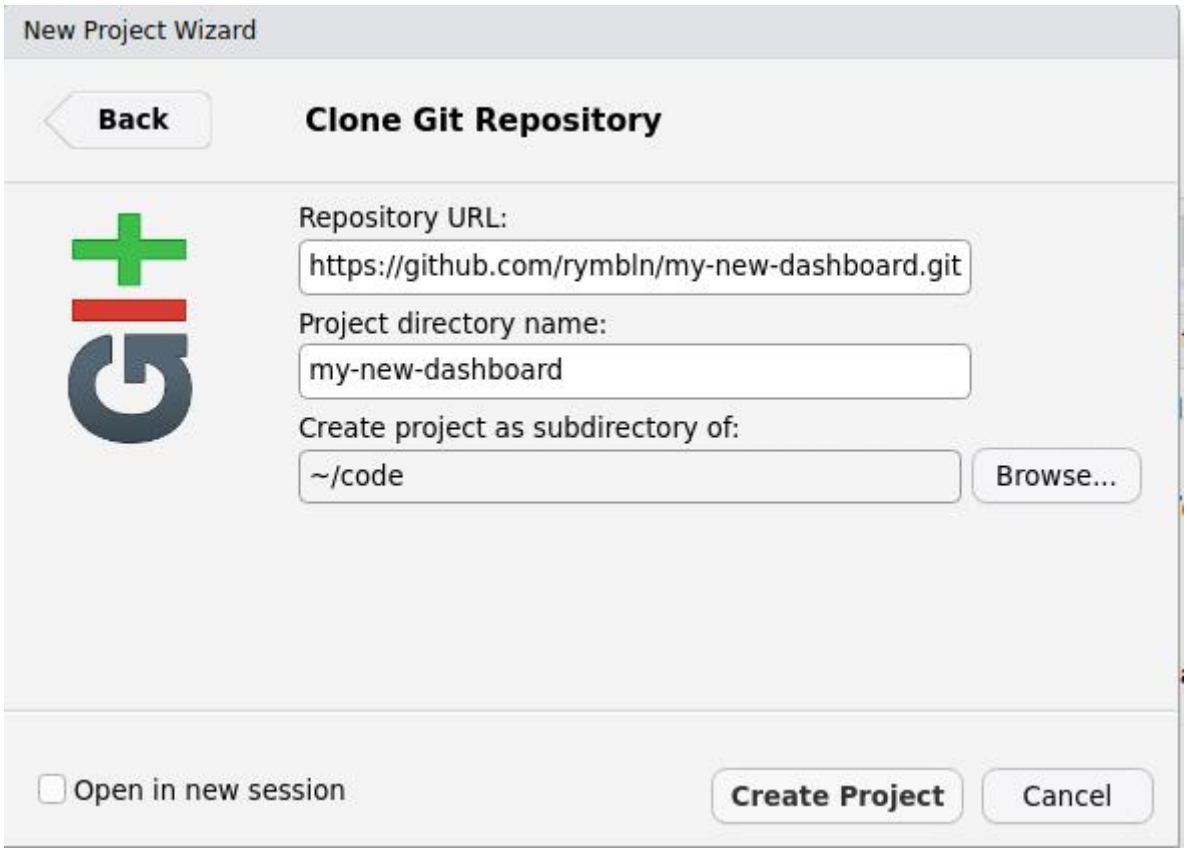

Рисунок 8.5. Клонирование репозитория (интерфейс Rstudio)

1. Когда появится вопрос про адрес репозитория “Repository URL”, необходимо вставить HTTPS URL из Github.
2. Далее необходимо присвоить проекту R короткое информативное имя.
3. Выбрать путь, где проект R будет сохранен локально.
4. Отметить поле *Open in new session* (открыть в новой сессии) и кликнуть *Create project* (создать проект).

Таким образом создается локальный проект RStudio, который является клоном удаленного репозитория GitHub.

Непосредственно клонировать репозиторий можно и с помощью команды терминала.

```
git clone https://github.com/rymbln/my-new-dashboard.git
```

ПОсле этого становится возможным открыть папку проекта в RStudio.

8.1.4. Подход RStudio-first

В случае использования подхода **RStudio-first** у исследователя уже есть проект в RStudio и необходимо создать для него репозиторий на GitHub.com. В этом сценарии прежде всего нужно создать локальный репозиторий в папке проекта с помощью команды

```
git init
```

При необходимости можно создать файл ***.gitignore***. Данный шаг можно пропустить, если при создании проекта уже была отмечена галочка *Create a git repository*. Далее необходимо создать первый коммит с файлами проекта.

```
# Добавление файлов проекта в индекс.
git add .
# Создание коммита с сообщением init.
git commit -m "init"
```

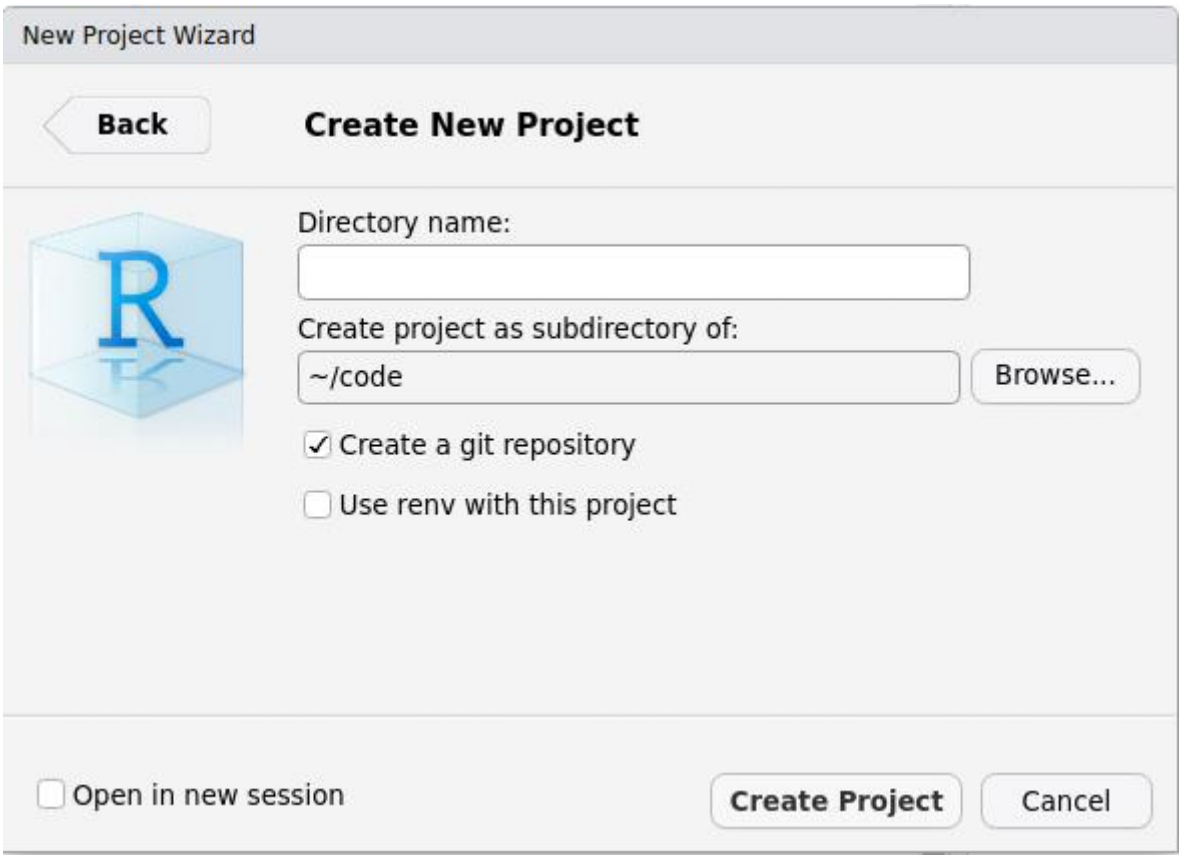

Рисунок 8.6. Создание проекта

Теперь необходимо создать новый репозиторий в GitHub. В этом случае можно не добавлять файлы ***.gitignore*** и ***README.md***. После создания репозитория на сайте будут показаны команды для выполнения дальнейших действий. Далее необходимо найти раздел “*push an existing repository from the command line”*. Следует переключить ссылку на репозиторий на режим https и скопировать ее.

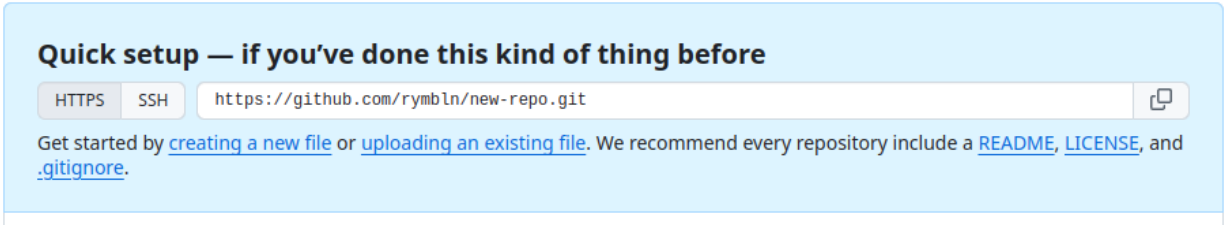

Рисунок 8.7. Копирование ссылки на существующий репозиторий

Далее необходимо последовательно выполнить в терминале команды, скопированные со страницы GitHub. Прежде всего следует добавить ссылку на удаленный репозиторий.

```
# Добавление ссылки на удаленный репозиторий.
git remote add origin https://github.com/rymbln/new-repo.git
# Создание ветки репозитория.
git branch -M main
# Отправление ветки репозитория на GitHub.
git push -u origin main
```

После этого, если обновить страницу репозитория в GitHub, можно увидеть файлы проекта (которые изначально были созданы локально) в удаленном репозитории.

8.1.5. Работа с Git в RStudio

Если в проект добавлен в репозитороий, то в RStudio появится новая вкладка *Git*.

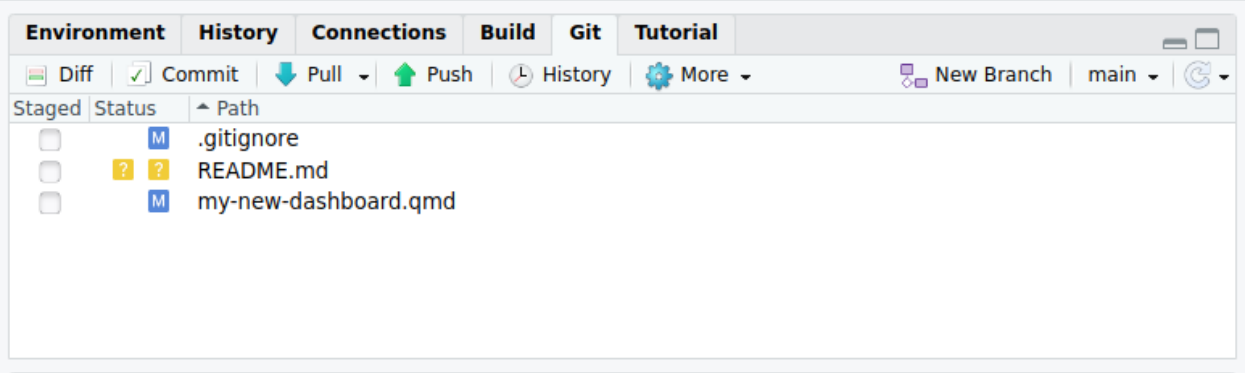

Рисунок 8.8. Вкладка Git в RStudio

Данная вкладка отображает статус файлов в репозитории и содержит кнопки для работы с ними:

- **Diff** - открывает окно просмотра изменений, которые произошли с файлами;
- **Commit** - открывает окно фиксации сохраненных изменений в отмеченных файлах;
- Синяя стрелка **Pull** - получает изменения для выбранной ветви из удаленного репозитория;

- Зеленая стрелка **Push** - отправляет зафиксированные изменения в удаленный репозиторий;
- **History** - открывает окно просмотра коммитов в репозитории;
- **New branch** - создает новую ветвь в репозитории на основе текущей;
- Между кнопкой **New branch** и закольцованной стрелкой расположен переключатель активной ветви в репозитории (на **Рисунке 8.8.** выбрана ветвь main).

Под кнопками отображаются добавленные, измененные и удаленные файлы из репозитория. Их статус кодируется цветом и специальным символом:

- A - Added - (бирюзовый) - файл добавлен в индекс;
- D - Deleted (красный) - файл удален из репозитория;
- M - Modified (синий) - файл изменен;
- R - Renamed (фиолетовый) - файл переименован;
- ? (желтый) - файл не включен в индекс.

Для выбора файлов, которые необходимо включить в индекс и создать коммит, их нужно отметить галочками.

Сценарий работы с Git и GitHub

После настройки получился удаленный репозиторий на GitHub и его локальная копия на компьютере. Поскольку теперь все изменения файлов фиксируются в коммитах, рабочий процесс может претерпеть некоторые изменения. Основная ветвь (созданная по умолчанию - master или main) — это так называемая “активная” версия всех файлов. Когда необходимо внести изменения, хорошей практикой будет создать новую ветвь из основной ветви. В новую ветвь вносятся изменения, а затем после завершения этапа работы она объединяется с основной.

Типичный рабочий процесс с репозиторием выглядит следующим образом:

1. Обновление локального репозитория, чтобы получить все изменения из удаленного (pull).
2. Переход в ветвь, над которой проводилась работа, либо создание новой ветви, чтобы что-либо протестировать (checkout).
3. Работа над файлами локально на своем компьютере, создание одного или нескольких коммитов в этой ветви (commit).
4. Обновление удаленной версии ветви, чтобы внести изменения из ветви локального репозитория (push).
5. Когда работа с ветвью завершена, ее можно объединить с основной ветвью или любой другой ветвью (merge).

Предполагается, что другие члены команды могут работать с репозиторием одновременно и данный процесс позволяет минимизировать конфликты. Схематично можно представить данный процесс следующим образом.

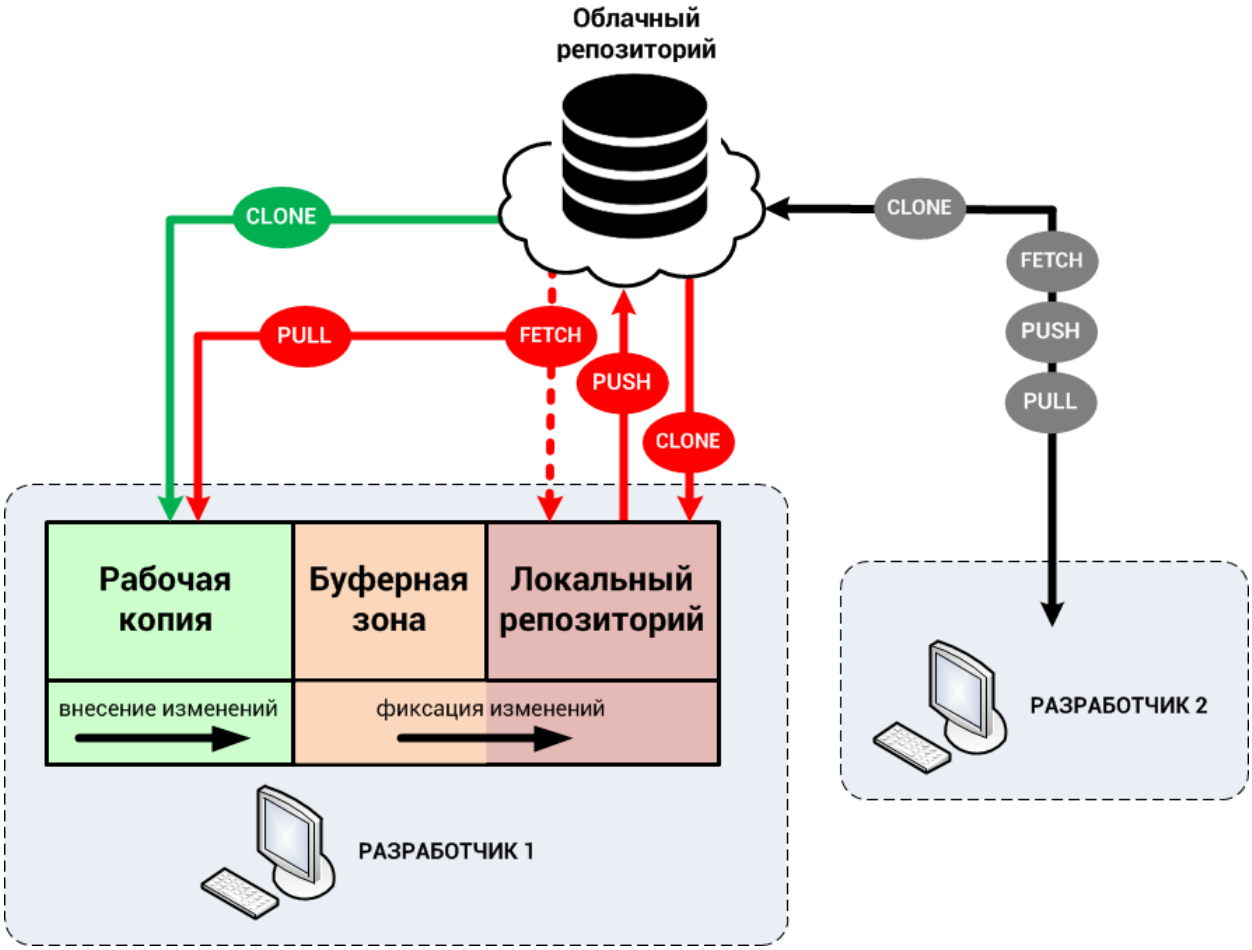

Рисунок 8.9. Схема взаимодействия двух разработчиков с помощью Git

Следует разобрать основные операции подробнее. Для примера будут использованы интерфейс RStudio и команды терминала.

Создание новой ветви

Прежде всего необходимо проверить, в какой ветке сейчас находится разработчик (можно увидеть на вкладке Git).

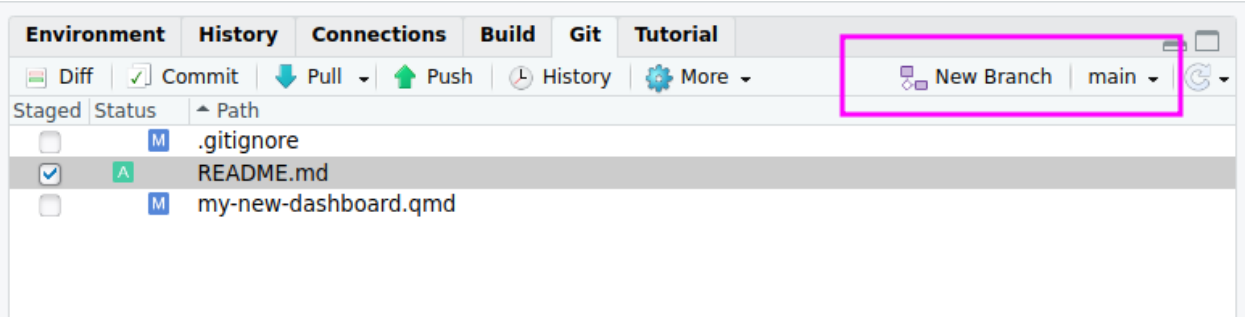

Рисунок 8.10. Обозначение ветви на вкладке Git в RStudio

Из **Рисунка 8.10.** видно, что сейчас работа происходит в основной ветке main. Далее можно создать новую ветку с помощью кнопки *New branch*.

При создании ветви нужно будет задать ее имя. Существуют общепринятые рекомендации по именованию веток:

- короткие, но описательные имена, чтобы можно было понять над чем ведется работа;
- делить слова в имени ветки дефисами или нижним подчеркиванием;
- использовать специальные префиксы, такие как **feature** или **fix** для разных типов веток (префиксы помогают быстро идентифицировать тип ветки и характер изменений);
- не использовать спец-символы @, <, >, $.

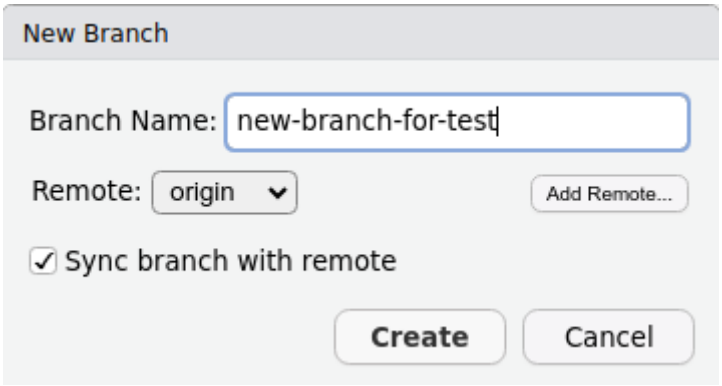

Рисунок 8.11. Создание новой ветви

При создании ветви можно отметить галочкой пункт *Sync branch with remote*, тогда ветвь будет создана не только в локальном, но и в удаленном репозитории.

После создания ветви, ее можно увидеть в панели *Git*. Здесь же можно переключаться между ветвями.

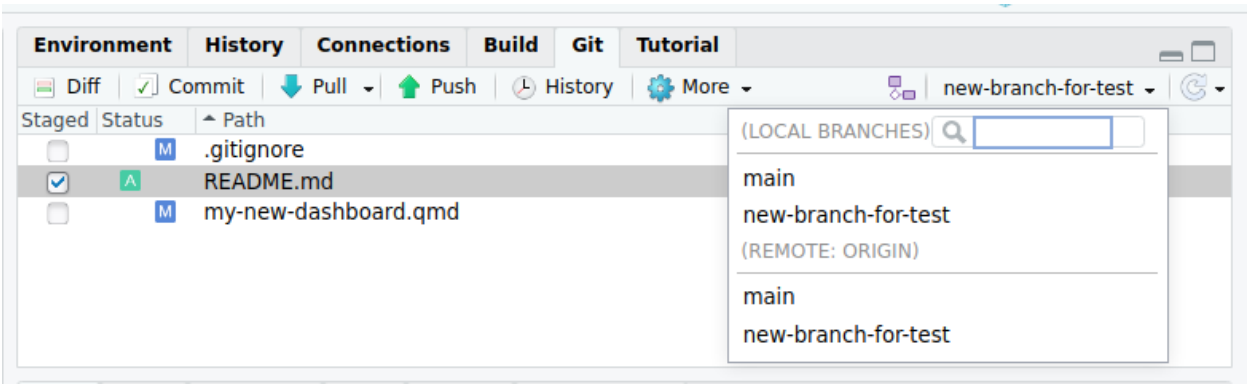

Рисунок 8.12. Переключение между ветвями в RStudio

Аналогичные действия можно сделать и с помощью команд терминала.

```
# Проверить текущий статус репозитория, в какой ветви находится.
```

```
git status
# Создание новой ветви.
git branch new-branch-for-test
# Переключение на новую ветвь.
git checkout new-branch-for-test
# Создание ветки и переключение на нее (одна команда для двух операций).
git checkout -b new-branch-for-test
# Просмотреть список ветвей.
git branch
```

Фиксация изменений

Теперь в новой ветви можно редактировать код, добавлять новые файлы, обновлять наборы данных и т.п. Каждое изменение отслеживается, как только соответствующий файл будет сохранен. Измененные файлы будут появляться во вкладке RStudio Git.

Рекомендуется значительные блоки изменений фиксировать — делать коммит. Как правило, лучше делать мелкий коммит, который легко откатить если возникнет проблема, чтобы одновременно сделать коммит по модификациям, объединенным общей целью. Чтобы этого добиться, нужно часто выполнять коммит. В начале внедрения практики использования системы Git исследователи и разработчики часто забывают делать коммит, но потом это входит в привычку.

Для каждого коммита нужно задать сообщение, которое будет описывать суть внесенных изменений. Для сообщений также есть некоторые рекомендации:

- Избегать односложных сообщений: «ок», «work», «wip», «fix» — они не информативны.
- Формулировать сообщение конкретно. Оно должно давать четкое представление о том, что было изменено и почему. Когда пишется сообщение, необходимо задать себе эти вопросы и записать краткий ответ в одно предложение.
- Избегать неоднозначности. В продолжение предыдущего пункта — лучше избегать нечетких формулировок: «исправление ошибок», «фикс багов» или «улучшения». Необходимо описывать конкретно то, что было сделано.
- Стараться укладываться в 72 символа. Сообщение должно оставаться кратким, чтобы его можно было читать без прокрутки. Если нужно более подробное описание, то после первой строки следует оставить пустую строку, а затем добавить подробности.

Пример ниже (**Рисунок 8.13.**) показывает, что с момента последнего коммита файлы ***.gitignore*** и ***my-new-dashboard.qmd*** были изменены, файл ***docs/logo.png*** удален, а файл ***README.md*** еще не индексируется.

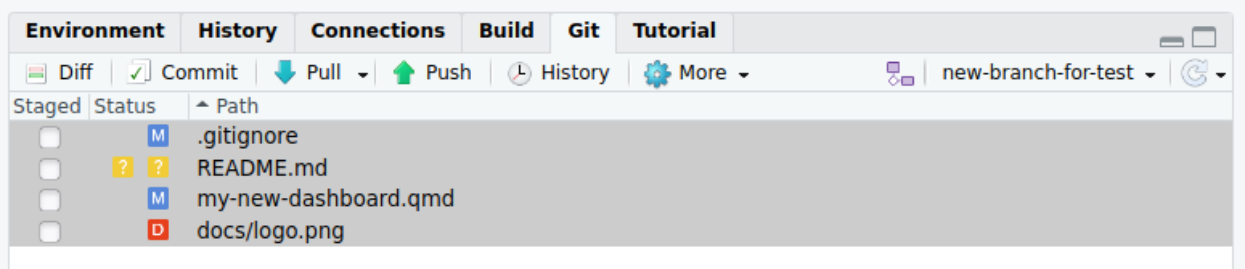

Рисунок 8.13. Пиктограммы состояния файлов с момента последнего коммита

Путь файла от создания до коммита можно представить следующим образом.

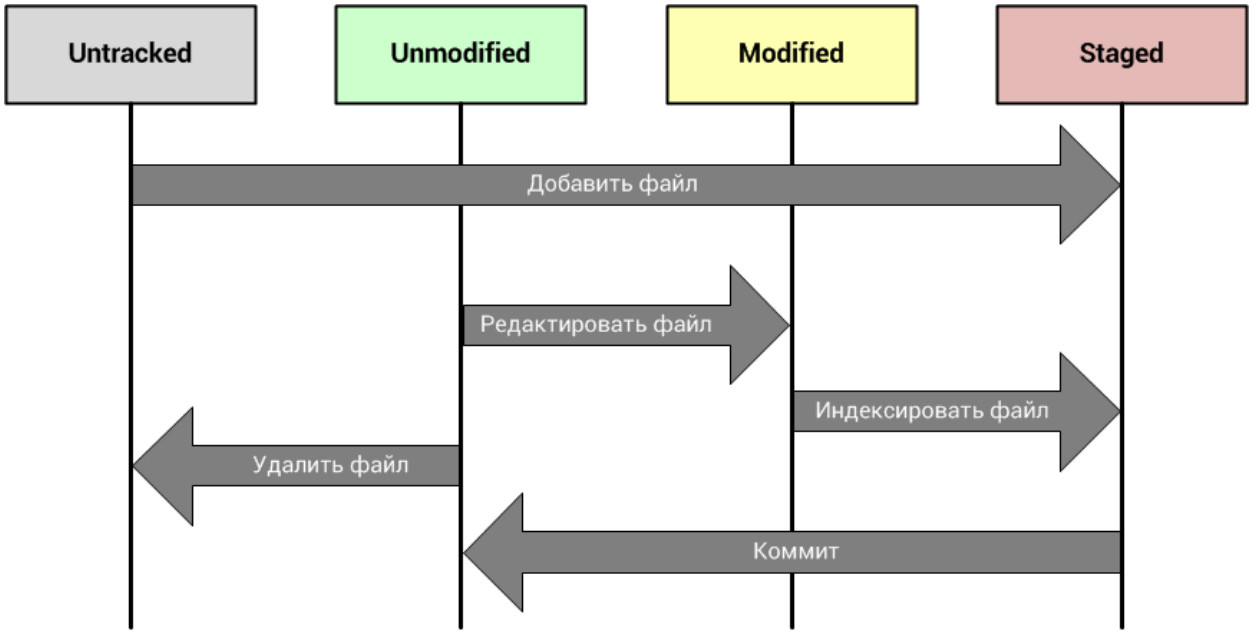

Рисунок 8.14. Путь файла от создания до коммита

Когда файл только создан, он еще не включен в репозиторий — он находится в состоянии **Untracked**. Чтобы включить его в репозиторий и начать отслеживание изменений, файл должен быть добавлен в индекс — тогда он будет в состоянии **Staged.** Все файлы добавленные в индекс можно фиксировать в коммите. После фиксации такие файлы считаются неизмененными — состояние **Unmodified**. Когда вносятся изменения в такой файл, он становится **Modified**. Эти изменения можно также включить в индекс — состояние **Staged** для последующей фиксации в коммите.

Для создания коммита следует нажать кнопку *Commit* в панели *Git*. Появится диалоговое окно (**Рисунок 8.15.**), в котором можно будет просмотреть состояния и изменения всех файлов проекта.

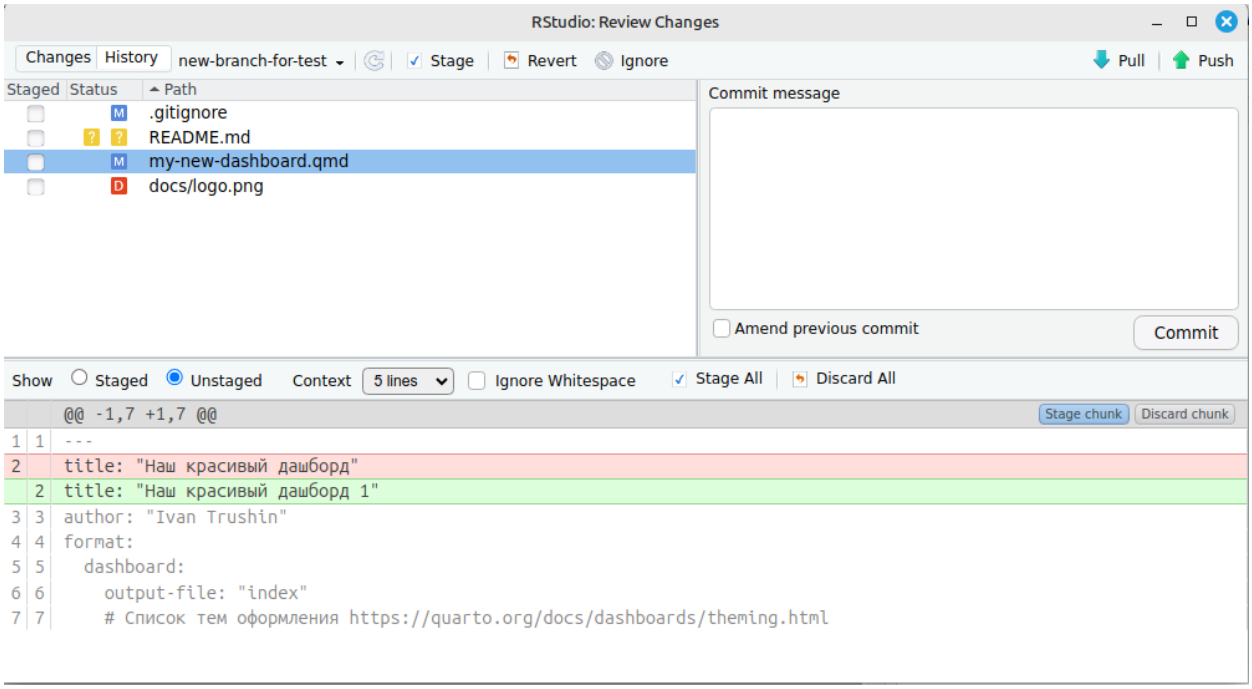

Рисунок 8.15. Состояния и изменения файлов проекта

Работа с диалоговым окном включает следующие основные моменты:

- Кликнув на имя файла в верхнем левом поле можно посмотреть изменения, которые были внесены в этот файл. Изменения отображаются в нижнем окне (выделено зеленым — то что было добавлено/после изменений; красным — то что было удалено/до изменений).
- Можно добавить файл в индекс (состояние **Staged**) отметив его галочкой или выделив все нужные файлы и нажав кнопку *Stage*.
- Дополнительно можно отменить изменения в файле до состояния последнего коммита с помощью кнопки *Revert* или добавить файл в список игнорирования **.gitignore** с помощью кнопки *Ignore*.
- В правом верхнем окне необходимо написать сообщение для коммита.
- При нажатии на кнопку *Commit* изменения будут зафиксированы и создан коммит. Появится всплывающее окно, показывающее сообщение о выполнении или ошибке.

Аналогичные действия можно проделать и в терминале. Состояние репозитория можно проверить с помощью команды **git status** - кроме состояния файлов она также покажет команды, для выполнения действий над этими файлами.

```
user@pc:~/code/my-new-dashboard$ git status
On branch new-branch-for-test
Your branch is up to date with 'origin/new-branch-for-test'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   my-new-dashboard.qmd
```

```
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   .gitignore
        deleted:    docs/logo.png

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        README.md
```

Для добавления файла в индекс используется команда ***git add <file>***. Можно добавлять по одному файлу, например ***git add my-new-dashboard.qmd***, можно сразу несколько. Чтобы добавить все файлы можно воспользоваться командой ***git add .***.

Для просмотра изменений в файле существует команда ***git diff <file>***.

```
rymbln@rymbln-N56VB:~/code/my-new-dashboard$ git diff my-new-dashboard.qmd
diff --git a/my-new-dashboard.qmd b/my-new-dashboard.qmd
index f3ccafe..a185e2f 100644
--- a/my-new-dashboard.qmd
+++ b/my-new-dashboard.qmd
@@ -1,5 +1,5 @@
 ---
-title: "Наш красивый дашборд"
+title: "Наш красивый дашборд 1"
 author: "Ivan Trushin"
 format:
   dashboard:
```

Чтобы отменить изменения в файле можно воспользоваться командой ***git restore <file>***.

После того, как все файлы для фиксации добавлены в индекс можно создать коммит с помощью команды ***git commit -m "сообщение коммита"***. Каждый коммит кодируется специальным хешем. Их можно увидеть в истории коммитов в RStudio.

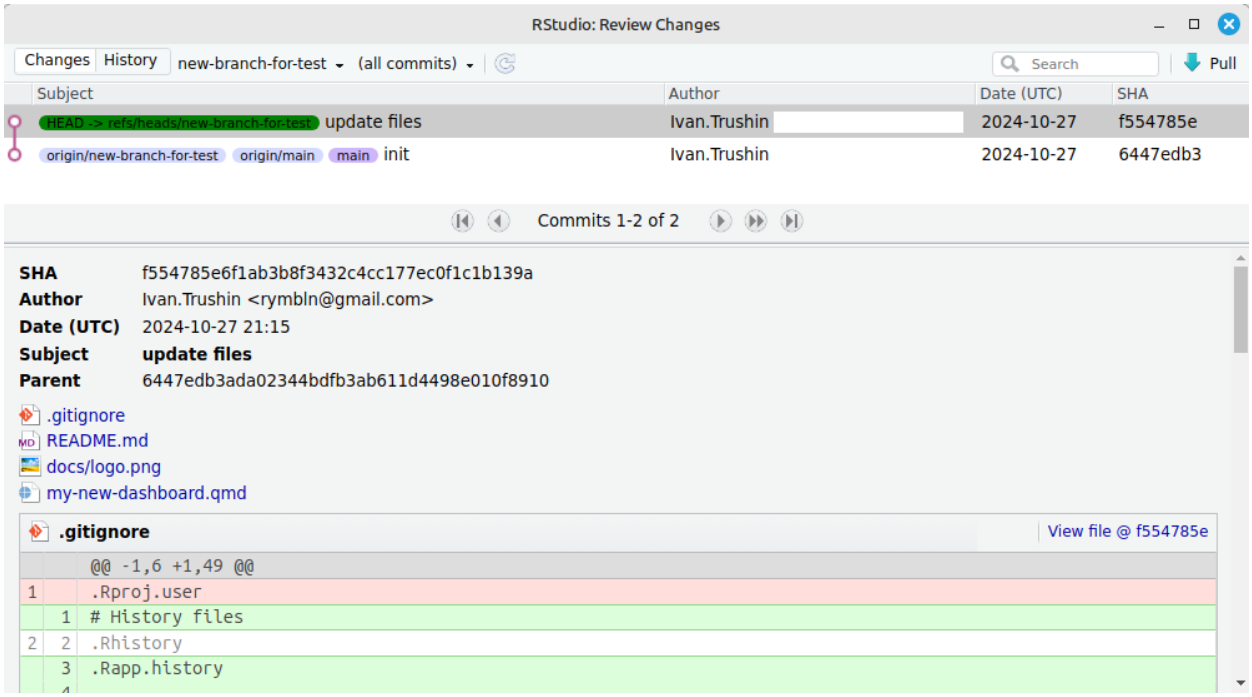

Рисунок 8.16. История коммитов в RStudio

Данные хеши можно использовать для перехода между коммитами с помощью команды ***git checkout***.

Внесение изменений в предыдущий коммит

Может возникнуть ситуация, что был создан коммит, однако возникла необходимость внести еще изменения, которые относятся к уже созданному коммиту. Можно "добавить" новые изменения к уже созданному коммиту. Для этого, при создании нового коммита в RStudio необходимо в диалоговом окне отметить галочкой пункт *Amend previous commit*.

В терминале аналогичное действие можно совершить командой

```
git commit --amend -m "сообщение нового коммита".
```

Рекомендуется вносить такие изменения в коммиты только в случае, если изменения еще не были отправлены в удаленный репозиторий. Гораздо легче откатить изменения, для которых был сделан коммит, но которые не были выгружены (т.е. все еще хранятся локально), чем откатить изменения, которые были переданы в удаленный репозиторий (и их уже кто-то скачал).

Отправка и получение изменений

Все создаваемые коммиты хранятся только локально на компьютере. Чтобы отправить их в удаленный репозиторий нужно совершить отдельное действие. В RStudio для этого существуют кнопки *Push* для отправки изменений и *Pull* для получения.

В терминале для этого служат аналогичные команды ***git push*** и ***git pull***. Стоит обратить внимание, что получение и отправка изменений осуществляется только для активной ветви (т.е. той ветви, над которой сейчас производится работа). Чтобы проверить, есть ли более свежие коммиты в удаленном репозитории без их непосредственного скачивания, можно воспользоваться командой ***git fetch*.**

При отправке и получении изменений в зависимости от настроек операционной системы может запрашиваться логин и пароль к удаленному репозиторию. Чтобы избежать этого можно настроить доступ к репозиторию по SSH-ключу. Способ настройки может отличаться в зависимости от сервиса для хранения удаленных репозиториев, поэтому за подробными настройками лучше обратиться в соответствующие справочные разделы GitHub.com, GitLab.com и т.д.

Возможна ситуация, когда были внесены изменения в локальном репозитории, но потом обнаруживается, что в удаленном репозитории есть коммиты, которые не были скачаны. В этом случае Git откажется их скачивать, так как это может привести к потере уже внесенных локальных изменений. Для решения такой ситуации может быть две стратеги:

- коммит локальных изменений, скачивание удаленных коммитов, разрешение конфликтов и последующая отправка совместных изменений;
- отложить свои изменения с помощью команды ***stash***, скачивание удаленных коммитов, восстановление изменений, фиксация, разрешение конфликтов и отправка совместных изменений.

Из интерфейса RStudio команду *stash* выполнить нельзя, но можно сделать через терминал. Команда ***git stash*** позволяет "спрятать" локальные изменения, а команда ***git stash pop*** "достать" их обратно.

Если файлы, к которым относятся удаленные изменения, и изменения в файлах на локальном компьютере не пересекаются, то Git может устранить конфликты автоматически.

Слияние ветвей

Если в ветвь не планируется больше вносить изменения, то можно начать процесс слияния этих изменений с основной веткой. В зависимости от ситуации, это может быть быстрым процессом, однако может потребоваться тщательная проверка и утверждение с участием членов команды.

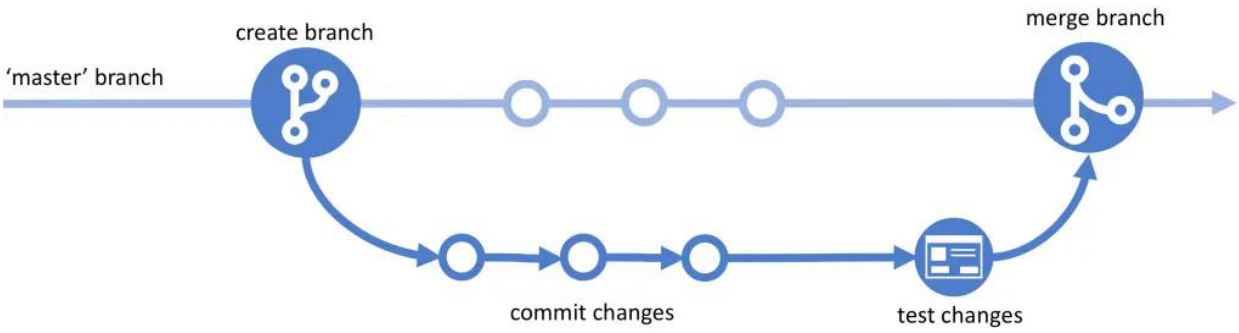

Рисунок 8.17. Схема ветвления и слияние ветвей

В интерфейсе RStudio нет специальных возможностей для слияния ветвей, поэтому все действия придется выполнять через терминал.

Сначала нужно перейти на ветвь, с которой будет осуществляться слияние. То есть это та ветвь, которая будет ПОЛУЧАТЕЛЕМ изменений. Это обычно ***master*** или ***main*** ветвь, но это может быть и другая ветвь. Затем необходимо провести слияние рабочей ветви с основной ветвью.

```
# Показать список ветвей.
user@pc:~/code/my-new-dashboard$ git branch
  main
* new-branch-for-test # Звездочкой отмечена ветвь, в которой мы находимся
# Перейти на ветвь main.
user@pc:~/code/my-new-dashboard$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
# Выполнить слияние с ветвью.
user@pc:~/code/my-new-dashboard$ git merge new-branch-for-test
Updating 6447edb..f554785
Fast-forward
 .gitignore           |  47 ++++++++++++++++++++++++++++++++++++++++++++++--
 README.md            |   2 ++
 docs/logo.png        | Bin 10938 -> 0 bytes
 my-new-dashboard.qmd |   2 +-
 4 files changed, 48 insertions(+), 3 deletions(-)
 create mode 100644 README.md
 delete mode 100644 docs/logo.png
```

В примере выше слияние прошло успешно, но так бывает не всегда. Если два пользователя модифицировали одну и ту же строку одновременно, возникает конфликт объединения. Также конфликты могут возникать при получении изменений из удаленного репозитория. Такие конфликты необходимо решать самостоятельно.

Разрешение конфликтов

В случае возникновения конфликта проблемные файлы будут помечены как CONFLICT, а расхождения в файле будут помечены следующим образом.

```
<<<<<<< HEAD
// Use a for loop to console.log contents. for(var i=0; i<arr.length; i++) {
console.log(arr[i]); }
=======
// Use forEach to console.log contents. arr.forEach(function(item) {
console.log(item); });
>>>>>>> Feature's commit.
```

Текст между `<<<<<<<` *HEAD* и `=======` идет из локального репозитория, а текст между `=======` и `>>>>>>>` — из другой ветви (которая может быть исходной, мастер или любой выбранной ветвью).

Необходимо решить, какую версию кода нужно оставить. Или вовсе написать третью версию кода и удалить все отметки, добавленные Git (`<<<<<<<` *HEAD*, `=======`, `>>>>>>>` *origin/master/your_branch_name*).

После этого можно сохранить файл, добавить в индекс и сделать коммит — этот коммит, который делает объединенные версии "официальными". Если слияние происходило при получении изменений, необходимо выгрузить полученный коммит в удаленный репозиторий. Чем чаще совместно работающие пользователи скачивают и выгружают, тем меньше будет вероятность конфликта. Как только ветвь объединена в мастер файл и больше не нужна, можно ее удалить с помощью команды *git branch -d new-branch-for-test*.

Создание тегов

Теги – это метки, предназначенные для обозначения важных коммитов. Они служат для пометки важных этапов работы, определения версии проекта, чтобы можно было быстро перейти на конкретное состояние проекта.

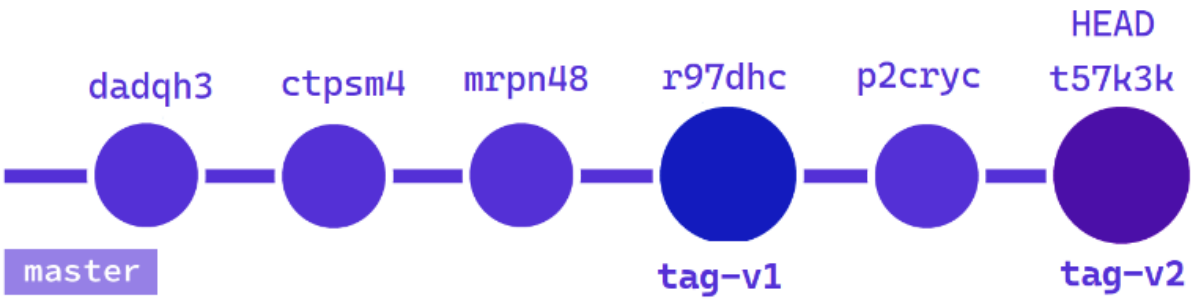

Рисунок 8.18. Цепочка тегов для коммитов

```
# Просмотр списка тегов.
git tag
```

```
# Переключение репозитория на конкретный тег.
git checkout <тег>
# Создание тега для последнего коммита.
git tag -a <тег>
# Создание тега для конкретного коммита.
git tag -a <тег> <хеш коммита>
# Удаление тега.
git tag -d <тег>
```

8.1.6. Дополнительные возможности GitHub

Как сервис хостинга для репозиториев, GitHub.com также предоставляет дополнительные возможности для работы, которые не относятся напрямую к технологии Git.

Форк (Fork)

Форк (от англ. fork – вилка) – точная копия репозитория, но скопированная в персональный аккаунт пользователя. Форки используются для создания независимой копии репозитория, в которую можно вносить изменения без влияния на основной репозиторий. Это позволяет пользователям вносить свой вклад в проект, не имея прав доступа к основному репозиторию, или экспериментировать с изменениями в безопасной среде. Данная функция позволяет создать собственную копию репозитория в своем профиле GitHub и модифицировать его для личного использования. При этом сохраняется связь с оригинальным репозиторием и при желании можно предложить интегрировать свои изменения с оригинальным репозиторием с помощью механизма *Pull Request*.

Запросы на слияние (Pull request)

Пулл-реквест (от англ. pull-request – pull-запрос) – функция GitHub, позволяющая попросить владельца репозитория, от которого сделан форк, загрузить пользовательские изменения обратно в свой репозиторий. На самом деле можно делать *Pull Request* и для своего репозитория — это также может быть одним из способов организации работы.

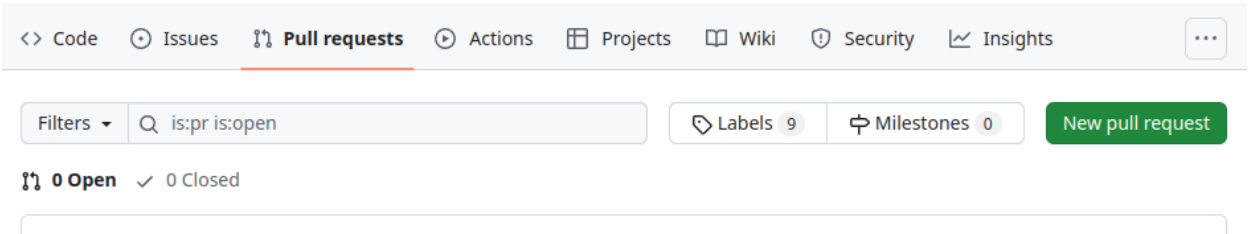

Рисунок 8.19. Страница GitHub с Pull Request

Обычно Pull Request создаются на основе отдельной ветви в репозитории. В рамках Pull Request удобно смотреть сразу все изменения, которые происходили с проектом в рамках этапа работы. Кроме того, всегда можно добавить коммиты к уже существующему пулл реквесту, просто добавив их к этой ветке в репозитории.

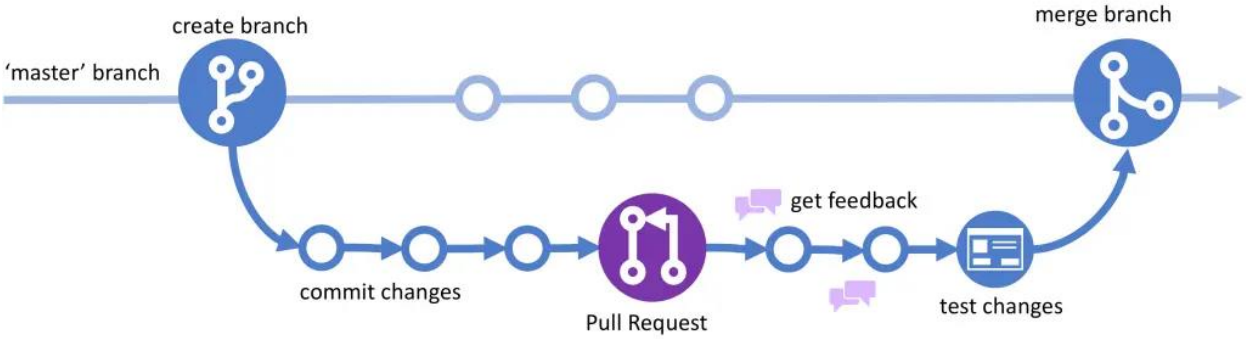

Рисунок 8.20. Схема Pull Request

При просмотре пулл реквеста кроме названия, описания и коммитов, также отображаются:

- комментарии, оставленные к пулл реквесту;
- дополнительные коммиты, добавленные к ветви пулл реквеста;
- комментарии к измененным строкам или файлам, оставленные к любому из коммитов, включенных в пулл реквест.

Предполагается, что Pull Request должен одобрять какой-либо отдельный пользователь, поэтому механизм Pull Request можно использовать как результат проверки совместной работы.

Github Actions

GitHub Actions — это платформа непрерывной интеграции и непрерывной поставки (CI/CD), которая позволяет автоматизировать конвейер сборки, тестирования и развертывания.

Благодаря ей можно создавать рабочие процессы для построения и тестирования различных приложений и запросов. Можно настроить запуск этих процессов на определенные события, например создание коммита, назначение тега и так далее.

Данные процессы выполняются либо на виртуальных машинах Linux, Windows и macOS, предоставляемых GitHub, либо можно настроить их запуск на собственных устройствах.

Подробнее работа с GitHub Actions будет рассмотрена в следующих разделах.

Github Pages

GitHub Pages — это бесплатный хостинг для статических файлов. Под статическими файлами подразумеваются сайты, для работы которых достаточно только файлов HTML, CSS, JS и не требуется какая-либо обработка со стороны сервера.

Изначально данный сервис разрабатывался как средство публикации документации для проектов, но затем перерос в нечто большее. Отдельным преимуществом является то, что GitHub Pages можно настроить таким образом, что удастся хранить в одном репозитории непосредственно код проекта и сайт для него. Подробнее работа с GitHub Pages будет рассмотрена в следующих разделах.