Git

Задание: Продвинутая работа с Git в команде

Цель задания:

Отработать навыки работы с Git, включая ветвление, разрешение конфликтов, управление тегами, выполнение `rebase`, настройку удалённого репозитория и подготовку релизов.

Сценарий задания:

Вы работаете над проектом в команде. Ваша задача: реализовать новую функциональность, разрешить конфликты, выполнить `rebase`, подготовить релиз и исправить критическую ошибку.

Шаги выполнения

1. Клонируйте репозиторий:

Создайте удалённый репозиторий или используйте готовый. Клонируйте его на локальный компьютер.

```
bash

git clone <url>
```

```
cd <папка_репозитория>
```

2. Настройка репозитория:

- Создайте файл `.gitignore` для исключения временных файлов.
- Добавьте `README.md` с кратким описанием проекта.
- Сделайте коммит.

```
bash

echo "node_modules/" > .gitignore
echo "# Project Name" > README.md
git add .gitignore README.md
git commit -m "Добавил .gitignore и README"
```

3. Создайте ветки для разработки:

- Создайте ветку `develop` для основной разработки.
- Создайте ветку `feature/add-api` для новой функциональности.

```
bash

git branch develop
git checkout develop
git branch feature/add-api
git checkout feature/add-api
```

4. Реализуйте функциональность в ветке `feature/add-api`:

- Создайте файл `api.js` и добавьте базовый код.

- Сделайте два коммита, чтобы зафиксировать изменения поэтапно.

```
bash

echo "console.log('API init');" > api.js
git add api.js
git commit -m "Добавил базовую структуру API"
echo "module.exports = {};" >> api.js
git commit -m "Добавил экспорт модуля API"
```

---

5. Создайте изменения в `develop`:

- Переключитесь на ветку `develop` и внесите изменения в файл `README.md`.
- Сделайте коммит.

```
bash

git checkout develop
echo "## Новая информация в README" >> README.md
git add README.md
git commit -m "Обновил README.md"
```

---

6. Сделайте `rebase` ветки `feature/add-api` на `develop`:

- Переключитесь обратно в ветку `feature/add-api` и выполните `rebase`.
- Убедитесь, что ваши изменения в `feature/add-api` появляются после изменений в `develop`.

```
bash

git checkout feature/add-api
```

```
git rebase develop
```

- Если в процессе `rebase` возникнут конфликты (например, если изменения затронули файл `README.md`), разрешите их вручную:

```
bash

# Отредактируйте конфликтные файлы
git add README.md
git rebase --continue
```

- После успешного выполнения `rebase`, отправьте ветку в удалённый репозиторий:

```
bash

git push origin feature/add-api --force
```

---

7. Подготовьте Pull Request:

- Создайте Pull Request для слияния ветки `feature/add-api` в `develop`.

---

8. Создайте релиз:

- После слияния изменений из `develop` в `main`, создайте тег версии `v1.0.0`.
- Запушьте тег в удалённый репозиторий.

```
bash

git checkout main
git merge develop
git tag -a v1.0.0 -m "Release version 1.0.0"
```

```
git push origin main --tags
```

---

9. Исправьте критическую ошибку:

- Создайте ветку `hotfix/fix-critical-bug` от ветки `main`.
- Внесите исправление в файл (например, добавьте обработку ошибки в `api.js`).
- Сделайте коммит и объедините изменения обратно в `main`.

```
bash

git checkout -b hotfix/fix-critical-bug main
echo "console.error('Critical bug fixed');" >> api.js
git add api.js
git commit -m "Исправил критическую ошибку"
git checkout main
git merge hotfix/fix-critical-bug
```

- Создайте тег для версии `v1.0.1` и запушьте его.

```
bash

git tag -a v1.0.1 -m "Critical bugfix release v1.0.1"
git push origin main --tags
```

---

10. Проверьте историю репозитория:

- Убедитесь, что изменения из `feature/add-api` появились после изменений из `develop`.
- Используйте следующую команду:

```
bash

git log --graph --oneline
```

Критерии оценки:

- Корректное использование `rebase` и разрешение конфликтов.
- Чистота истории коммитов после выполнения `rebase`.
- Правильное создание и управление ветками.
- Работа с тегами и подготовка релиза.
- Корректное исправление ошибок и работа с веткой `hotfix`.

Дополнительные задания:

1. Проверьте разницу между `merge` и `rebase` на этом же проекте.
2. Используйте интерактивный `rebase` для редактирования истории (`git rebase -i`).
3. Внесите изменения в `README.md` одновременно в ветках `develop` и `hotfix/fix-critical-bug`, создайте конфликт и разрешите его.

Дополнительные задания:

---

1. Проверьте разницу между `merge` и `rebase`:

1. Создайте дополнительную ветку `feature/merge-vs-rebase` от `develop`:

```
bash

git checkout develop
```

```
git branch feature/merge-vs-rebase
git checkout feature/merge-vs-rebase
```

2. Внесите изменения в файл `README.md` в ветке `develop` (например, добавьте новую строку):

```
bash

git checkout develop
echo "Дополнительная информация о проекте." >> README.md
git add README.md
git commit -m "Обновил README.md в develop"
```

3. Вернитесь в ветку `feature/merge-vs-rebase` и внесите свои изменения:

```
bash

git checkout feature/merge-vs-rebase
echo "Описание нового функционала." >> README.md
git add README.md
git commit -m "Добавил описание нового функционала в READM
E"
```

4. Выполните сначала `merge` ветки `develop` в `feature/merge-vs-rebase`:

```
bash

git merge develop
```

- Разрешите возможные конфликты, сделайте коммит.

5. Удалите последний коммит слияния (`git reset --hard`) и выполните `rebase` вместо `merge`:

```
bash

git reset --hard HEAD~1
git rebase develop
```

6. Сравните историю после `merge` и `rebase`, используя команду:

```
bash

git log --graph --oneline
```

Вывод:

- После `merge` видна точка слияния, а после `rebase` история линейна.

---

2. Используйте интерактивный `rebase` для редактирования истории (`git rebase -i`):

1. Переключитесь на ветку `feature/add-api` и выполните несколько коммитов:

```
bash

echo "Добавил новый модуль" >> api.js
git add api.js
git commit -m "Добавил модуль API"
echo "Добавил документацию для API" >> api.js
git add api.js
git commit -m "Добавил документацию"
```

2. Выполните интерактивный `rebase` для редактирования истории:

```
bash

git rebase -i HEAD~2
```

- В редакторе измените команды для первого коммита (`pick` → `reword`) и для второго (`pick` → `squash`).
- После этого обновите сообщение коммита.

Результат:

- Один объединённый коммит с обновлённым описанием.

---

3. Разрешите конфликт в файле `README.md` одновременно в ветках `develop` и `hotfix`:

1. Внесите изменения в файл `README.md` в ветке `develop`:

```
bash

git checkout develop
echo "Дополнение из ветки develop." >> README.md
git add README.md
git commit -m "Обновил README.md из develop"
```

2. Переключитесь на ветку `hotfix/fix-critical-bug` и внесите в `README.md` другие изменения:

```
bash

git checkout -b hotfix/fix-critical-bug main
echo "Исправление из ветки hotfix." >> README.md
git add README.md
```

```
git commit -m "Обновил README.md из hotfix"
```

3. Попробуйте смержить `hotfix/fix-critical-bug` в `develop`:

```
bash

git checkout develop
git merge hotfix/fix-critical-bug
```

4. Разрешите конфликт вручную:
   - Откройте файл `README.md` и объедините изменения.
   - Сделайте коммит после разрешения конфликта:

```
bash

git add README.md
git commit -m "Разрешил конфликт в README.md между hotfix
и develop"
```

---

4. Проверьте структуру истории:

1. Выполните команду для визуализации истории ветвлений и слияний:

```
bash

git log --graph --oneline --all
```

2. Убедитесь, что:
   - `feature/add-api` правильно ребейзится на `develop`.

- `hotfix/fix-critical-bug` был объединён с `develop`.
- История выглядит линейно после применения `rebase`.

Критерии оценки:

- Успешное выполнение всех шагов основного задания.
- Правильное выполнение `merge` и `rebase` с анализом различий.
- Корректное использование интерактивного `rebase` для редактирования истории.
- Успешное разрешение конфликтов в `README.md`.
- Чистота и логичность истории в репозитории.

Merge and Rebase

Разница между командами `git merge` и `git rebase` заключается в том, как они объединяют изменения из одной ветки в другую, что влияет на историю коммитов. Рассмотрим их подробнее:

1. `git merge`

- **Что делает:**

  Объединяет две ветки, создавая новый коммит слияния (merge commit). История каждой ветки сохраняется, даже если она разветвлена.

- **Особенности:**
  - История коммитов остаётся нетронутой.
  - Добавляется дополнительный коммит (merge commit), который отображает момент слияния.
  - Удобно для сохранения контекста работы в ветке.

- **Пример:**

  Если у вас есть ветки `main` и `feature`, и вы выполняете:

  ```
  bash

  git checkout main
  git merge feature
  ```

  Git создаст новый коммит, объединяющий изменения из обеих веток.

- **Визуализация:**

  ```
  mathematica

  A---B---C (main)
       \
        D---E (feature)
  После merge:
  A---B---C---F (main)
       \       /
        D---E (feature)
  ```

  (Где `F` — это merge-коммит.)

---

2. `git rebase`

- **Что делает:**

  Переносит коммиты из одной ветки поверх другой, изменяя историю коммитов.

- **Особенности:**
  - История становится линейной.
  - Оригинальные коммиты переписываются (создаются новые SHA-1 хэши).

- Используется, чтобы упростить историю коммитов.
- Требует осторожности при совместной работе (особенно с уже опубликованными ветками).

- **Пример:**

  Если вы выполняете:

  ```
  bash

  git checkout feature
  git rebase main
  ```

  Git перенесёт коммиты из ветки `feature` поверх текущей версии `main`.

- **Визуализация:**

  До rebase:

  ```
  css

  A---B---C (main)
       \
        D---E (feature)
  ```

  После rebase:

  ```
  css

  A---B---C---D'---E' (feature)
  ```

Когда использовать:

- `merge`:

- Когда важно сохранить историю веток.
- Для командной работы, чтобы понять, как и когда изменения были объединены.

- `rebase`:
  - Для создания чистой линейной истории, особенно при работе с небольшими ветками.
  - Перед публикацией изменений, чтобы сделать историю аккуратнее.

Совет:

Если вы работаете в команде:

- Никогда не используйте `rebase` для веток, которые уже были отправлены в общий репозиторий. Это может привести к конфликтам и путанице.
- Используйте `merge` для интеграции с основной веткой, чтобы сохранить контекст.

`git tag -a`

Команда `git tag -a` используется для создания аннотированных тегов в Git. Теги в Git применяются для маркировки конкретных точек в истории коммитов, например, для отметки выпусков (релизов) или других важных событий.

Аннотированные теги (annotated tags) содержат дополнительную информацию, такую как:

- Имя автора.
- Временная метка (дата и время создания).
- Сообщение, описывающее назначение тега.

- Криптографическая подпись (если настроена GPG-подпись).

Синтаксис

```
bash

git tag -a <имя_тега> -m "<сообщение>"
```

- `<имя_тега>` — имя, которое вы хотите присвоить тегу.
- `m "<сообщение>"` — сообщение, описывающее тег.

Пример

Допустим, вы хотите пометить текущий коммит как версию 1.0.0:

```
bash

git tag -a v1.0.0 -m "Release version 1.0.0"
```

Чтобы просмотреть список всех тегов, выполните:

```
bash

git tag
```

Если вы хотите увидеть подробную информацию о теге:

```
bash

git show v1.0.0
```

Отличие аннотированных тегов от лёгких (lightweight)

- **Аннотированные теги** хранятся как полноценные объекты в базе данных Git и содержат дополнительную информацию.
- **Лёгкие теги** — это просто указатель на конкретный коммит, без метаданных. Создаются без флага `a` или `m`:

```
bash

git tag v1.0.0
```

Как передать теги в удалённый репозиторий?

По умолчанию теги не отправляются при `git push`. Чтобы отправить аннотированный тег, используйте:

```
bash

git push origin v1.0.0
```

Для отправки всех тегов:

```
bash

git push origin --tags
```

Аннотированные теги чаще используются для релизов, так как они дают больше информации, чем лёгкие теги.

Лёгкие теги

Лёгкие теги (lightweight tags) в Git являются простыми указателями на коммит и не содержат дополнительных данных. Вот чего именно в них **нет** по сравнению с аннотированными тегами:

1. **Информация об авторе тега:**
   - В аннотированном теге сохраняется имя автора, создавшего тег.
   - В лёгком теге такой информации нет.
2. **Временная метка:**
   - Аннотированный тег включает дату и время создания тега.
   - Лёгкий тег не содержит временной метки — он просто указывает на коммит.
3. **Сообщение:**
   - Аннотированные теги позволяют добавлять сообщение (`m "<сообщение>"`), которое объясняет назначение тега.
   - В лёгком теге сообщение отсутствует.
4. **Криптографическая подпись:**
   - Аннотированные теги могут быть подписаны с помощью GPG для обеспечения аутентичности.
   - Лёгкие теги не поддерживают подпись.
5. **Объект в базе данных Git:**
   - Аннотированные теги создаются как отдельные объекты в базе данных Git (`git objects`), что позволяет хранить их метаданные.
   - Лёгкие теги — это просто ссылка на хеш коммита и не создают нового объекта в базе данных.

Вывод

Лёгкие теги удобны для быстрого создания и локальных пометок, но они лишены метаданных и дополнительных возможностей, которые делают аннотированные теги предпочтительными для важных событий, таких как релизы.