Использование Git при индивидуальной и командной разработке

Контрольный список:
- Инициализация нового репозитория Git и клонирование существующего.
- Сохранение изменений с помощью коммитов и просмотр истории.
- Создание и переключение между ветками при разработке функций и исправлении багов.
- Подключение удалённых репозиториев (GitHub/GitLab), команда `git push` / `pull`.
- Слияние веток, разрешение конфликтов и работа через Pull Request.
- Настройка CI/CD-процессов для автоматической сборки и тестирования.
- Настройка прав доступа и защита веток в командной разработке.

Создание и структура репозитория

Репозиторий Git — это папка с проектом, где ведётся учёт всех изменений. Чтобы начать пользоваться Git, в новом проекте выполняют:

```
mkdir myproject
cd myproject
git init            # Создаём новый локальный репозиторий Git
```

Это создаёт скрытую папку `.git`, где хранится вся история. Далее в проекте обычно создают файлы (например, `README.md`, код, `.gitignore` для исключения временных файлов) и сохраняют их в Git. Для этого изменения добавляют в индекс и делают коммит:

```
echo "# My Project" >> README.md
git add README.md   # Добавляем файл в индекс (готовим к коммиту)
git commit -m "Initial commit: create repository"  # Фиксируем снимок
состояния с сообщением
```

Сообщение коммита должно быть понятным (обычно в повелительном наклонении, коротко описывать изменения). Если есть готовый удалённый репозиторий (на GitHub, GitLab и т.д.), можно склонировать его одним шагом:

```
git clone https://github.com/user/myproject.git  # Клонируем удалённый
репозиторий
```

При клонировании Git автоматически создаёт копию проекта с настройками удалённого `origin`.

Работа с коммитами и историей изменений

Коммит (commit) — это «снимок» проекта в определённый момент. После изменений в файлах выполняют:

```
git add file.py     # Добавляем изменённый файл в индекс
git commit -m "Add file.py: implement initial function"
```

Это сохраняет текущие изменения с понятным сообщением. Чтобы посмотреть историю изменений, используют:

```
git log --oneline
```

где выводятся короткие записи коммитов:

```
a1b2c3d Add file.py: implement initial function
0a1b2c3 Initial commit: create repository
```

Команда `git status` помогает увидеть, какие файлы изменены, добавлены в индекс или ещё не отслеживаются. Также полезны команды `git diff` для просмотра различий перед коммитом. Важный момент — писать сообщения так, чтобы их было легко понять (например, `Fix login bug` или `Update README with instructions`). Следуя промышленному стандарту, обычно делают короткое заголовок (до ~50 символов) в описательной форме и, при необходимости, более подробное описание в теле сообщения.

Ветки и ветвление

Ветка (branch) — это параллельная версия проекта. Главная ветка обычно называется `main` (или `master`), где хранятся стабильные релизы. Для разработки новой функции или исправления создают отдельную ветку:

```
git branch feature/new-login   # Создаём ветку для новой функции
git checkout feature/new-login # Переключаемся на неё
```

Или с помощью одной команды:

```
git switch -c feature/new-login  # Создаёт и переключает на ветку new-login
```

Теперь все коммиты будут в ветке `feature/new-login`, не затрагивая `main`. После завершения работы ветку сливают обратно в `main`. Принято называть ветки осмысленно: например, `feature/*` для функций (`feature/add-auth`), `bugfix/*` для исправлений (`bugfix/fix-crash`), `hotfix/*` для срочных правок на продакшене. В названиях используют только строчные буквы и дефисы, без пробелов. Это облегчает командную работу: каждый понимает по названию, над чем идёт работа. Чтобы увидеть существующие ветки, используют

`git branch`; переключение между ними — `git checkout` или `git switch`. При работе в ветке регулярно делают коммиты:

```
# Ветка feature/new-login: добавляем форму логина
git add login_form.py
git commit -m "Add login form UI component"
```

Когда работа готова, нужно слить ветку с `main`: сначала перейти в `main`, подтянуть из удалённого (чтобы быть в актуальной версии), затем:

```
git checkout main
git pull origin main        # Обновляем локальную ветку main из удалённого
git merge feature/new-login # Сливаем изменения из feature/new-login в main
```

Если другие изменили `main` параллельно, возможно придётся решать конфликты (см. ниже). Если всё прошло гладко, делают коммит слияния. Затем новую функцию проверяют и удаляют ненужную ветку:

```
git branch -d feature/new-login  # Удаляем локально ветку после слияния
```

Такая стратегия (каждая новая фича в отдельной ветке) позволяет параллельно работать нескольким разработчикам, не мешая стабильной версии в `main`.

Удалённые репозитории и работа с ними

Удалённый репозиторий (remote) — это копия проекта на сервере (например, GitHub, GitLab). С ним связана ссылка `origin` по умолчанию. Чтобы отправить локальные коммиты в удалённый репозиторий, используют `git push`:

```
git push origin main  # Отправляем локальную ветку main в репозиторий origin
```

Если вы создали новую ветку и хотите загрузить её в удалённый репозиторий, делайте:

```
git push -u origin feature/new-login  # Отправляем новую ветку и
устанавливаем слежение
```

Флаг `-u` связывает локальную ветку с удалённой (будущие `git push/pull` без указания будут знать, куда идти). Чтобы получить обновления от других, используют `git pull`:

```
git pull origin main  # Получаем и сразу сливаем изменения из remote/main
```

`git pull` сочетает в себе `git fetch` (забирает изменения) и `git merge` (сливает их). Или можно вручную:

```
git fetch origin  # Забираем все ветки из origin без автоматического слияния
git merge origin/main  # Вливаем изменения из origin/main в текущую ветку
```

Например, при командной разработке каждый разработчик клонирует репозиторий через `git clone` и вносит правки локально, после чего делает `git push` своих изменений, чтобы другие могли их увидеть. Перед началом работы полезно делать `git pull`, чтобы убедиться, что локальная копия обновлена и нет конфликтов с чужими изменениями.

Слияние, разрешение конфликтов и Pull Request

Когда работа над фичей завершена, её нужно слить (merge) в основную ветку. Если изменений немного и они не перекрываются, Git выполнит слияние автоматически:

```
git checkout main
git merge feature/new-login  # Сливаем feature-ветку в main
```

Если же в одной и той же части файла работало несколько человек, возникает конфликт — Git помечает места вроде:

```
<<<<<<< HEAD
текст из ветки main
=======
текст из feature/new-login
>>>>>>> feature/new-login
```

Чтобы решить конфликт, откройте файл в редакторе, выберите правильную версию кода (или объедините их), удалите маркеры `<<<<<<<`, `=======`, `>>>>>>>`, затем сохраните и снова закоммитьте:

```
git add conflicted_file.py
git commit -m "Resolve conflict between main and feature/new-login"
```

После этого изменения будут слиты.

В командной разработке часто используют систему Pull Request (или Merge Request): вы отправляете запрос на слияние вашей ветки в `main` через интерфейс GitHub/GitLab. Это позволяет другим членам команды просмотреть изменения, обсудить их и убедиться, что код качественный. Обычно в настройках проекта защищают ветку `main`, запрещая прямые пуши в неё и требуя одобрения (review) и прохождения автоматических проверок (см. ниже) перед слиянием. Таким образом соблюдаются стандарты: каждый pull request должен содержать осмысленное описание, проходить код-ревью и успешно проходить тесты перед объединением в основную ветку.

Основы CI/CD и автоматическая проверка кода

CI/CD (Continuous Integration/Continuous Delivery) — это практика, при которой после каждого пуша кода автоматически запускаются сборка и тесты. Например, при наличии конфигурации в `.github/workflows/ci.yml` или `.gitlab-ci.yml` система автоматически проверит ваш код: соберёт проект, запустит тесты и статический анализ. Если всё прошло успешно, у девелопера (или в pull request) будет зелёный статус, и только после этого можно сливать ветку в `main`. Если тесты падают, Git пометит задачу как проваленную, и нужно исправить ошибки, прежде чем продолжать. Такой подход помогает не пропускать баги в основную ветку и ускоряет обратную связь при командной разработке.

Управление доступом и защита веток в команде

В командной работе важно управлять правами доступа к репозиторию. На платформах вроде GitHub можно назначать роли: кто может только читать, кто писать, кто администрировать репозиторий. Обычно главный разработчик или команда лидов создают проект и приглашают участников с нужными правами. Часто вводят правила защиты ветки `main` (branch protection): например, запрещают прямые `git push` в `main`, требуют прохождения CI и хотя бы одного одобрения перед слиянием. Такие правила повышают стабильность: код попадает в `main` только после проверки и обсуждения. Нередко создают несколько сред (веток) разработки, например, `develop` для интеграции фич перед релизом, а в `main` держат только полностью протестированный код. При обнаружении серьёзного бага создают горячую ветку (`hotfix/*`), быстро фиксируют её, а затем сливают и в `main`, и (если используется `develop`) обратно в неё. Благодаря правильным настройкам доступа и стандартам рабочего процесса команда работает слаженно и снижает риск ошибок.

Вопросы для самопроверки

- Как и зачем инициализировать новый репозиторий Git в проекте?
- В чём разница между `git add`, `git commit` и `git push`?
- Как создать новую ветку, переключиться на неё и затем вернуть изменения в основную ветку?
- Что такое конфликт при слиянии и как его разрешить?
- Зачем нужен удалённый репозиторий (GitHub/GitLab) и как работать с командами `git clone`, `git pull` и `git push`?
- Что такое Pull Request и какие преимущества он даёт в командной разработке?
- Какие задачи выполняют CI/CD, и как они помогают улучшить качество кода?
- Как в GitHub/GitLab настроить права доступа и защиту ветки `main`?