Неделя 8. Лекция 4. Git и GitHub: как команда работает с кодом

Часть 2 из 2. Начало лекции — в материалах предыдущей недели.

6. GitHub Flow: процесс команды

1. **Issue** описывает задачу: что и зачем.
2. **Ветка** от свежего `main`: `feat/3-rename-project`, `fix/15-null-ref`. Номер issue в имени связывает ветку с задачей.
3. **Маленькие коммиты** с понятными сообщениями.
4. **Pull Request** с `Closes #3`: при merge issue закроется сам.
5. **CI зелёный + ревью** → merge (обычно squash: вся ветка — один коммит в `main`).
6. `main` **автоматически деплоится** (тема 5).

Зачем: в `main` попадает только проверенный и просмотренный код, а у каждого изменения есть причина (issue) и обсуждение (PR).

Связь issue ↔ PR ↔ коммит. Ключевые слова `Closes`, `Fixes`, `Resolves` с номером (`Closes #3`) в описании PR закрывают issue при merge в ветку по умолчанию. Squash-коммит несёт номер PR, PR ссылается на issue. По одной строке `git log` через год можно дойти до обсуждения и понять не только *что* поменяли, но и *почему*.

Почему GitHub Flow, а не другие схемы:

Схема	Как устроена	Когда подходит
Trunk-based	Все вливают в `main` очень часто, ветки живут часы; незаконченное прячут за флагами	Зрелая команда с сильными автотестами
GitHub Flow	Короткие ветки от `main`, PR с ревью, `main` всегда готов к деплою	Сервис, который деплоится непрерывно, — наш случай
Git Flow	Ветки `develop`, `release/*`, `hotfix/*`, релизы по расписанию	Продукты с несколькими поддерживаемыми версиями, коробочные поставки

Сервис задач один, деплоится из `main` и не поддерживает старые версии параллельно. Git Flow добавил бы ветки, которые нечем оправдать, а trunk-based без обязательного ревью не учит главному, что мы отрабатываем в лабе.

Зачем маленькие PR. Ревьюер держит в голове ограниченный объём изменений. PR на 100 строк читают внимательно и находят ошибки, PR на 2000 строк пролистывают и одобряют. Маленький PR быстрее проходит ревью, реже конфликтует и проще откатывается. Если задача большая, её режут: сначала метод репозитория с тестами, потом эндпоинт.

Сообщения коммитов — Conventional Commits:

```
feat: статистика задач проекта
fix: 404 вместо 500 для несуществующего проекта
ci: публиковать образ в GHCR
docs: инструкция запуска
```

По префиксу видно, что меняет поведение, а история читается как список изменений. Хорошее сообщение отвечает на вопрос «что станет иначе после этого коммита», а не «что я делал».

```
flowchart LR
  I["Issue #3<br/>переименование"] --> B["ветка<br/>feat/3-rename-project"]
  B --> P["PR: Closes #3"]
  P --> R{"CI + ревью"}
  R -->|"замечания"| B
  R -->|"Approve"| M["squash merge<br/>в main"]
  M --> D["issue закрыт<br/>деплой"]
```

7. Pull Request и ревью кода

Pull Request (PR) — запрос «влейте мою ветку в `main`». В нём видны изменения построчно, обсуждение, статусы проверок CI.

Описание PR пишут для ревьюера, который не сидел рядом с вами. Шаблон `.github/pull_request_template.md` задаёт структуру: *что сделано*, `Closes #`, *как проверить* (compose поднимается, `/health/ready` = 200, `dotnet test` зелёный) и чек-лист про секреты и документацию. Описание «сделал задачу» бесполезно. Полезное — «Добавлен `PUT /api/projects/{id}`: 200, 400 на пустое имя, 404, 409 на занятое. Метод `RenameAsync` в обоих репозиториях, два интеграционных теста. Проверить: запрос из `DemoApi.http`».

Ревью — второй человек читает код до merge. Он ищет не опечатки, а проблемы:

Что смотреть	Пример в нашем проекте
Слои	Нет ли SQL в контроллере; правила — в `Domain`, запросы к БД — в `Data` (лекция 1)
Оба хранилища	Новый метод репозитория реализован и в памяти, и в PostgreSQL
Тесты	Есть тест на успех и на ошибку
Безопасность	Параметры в SQL, нет паролей и токенов
Эксплуатация	Не сломаны health checks, логи, настройки через окружение

Ревьюер оставляет комментарии к строкам, автор исправляет новыми коммитами, ревьюер ставит **Approve**. Если замечания серьёзные — **Request changes**, и PR не вольют, пока их не закроют.

Как писать комментарий. Комментарий — про код, а не про автора, и из него понятно, что сделать и почему.

Плохо	Хорошо
«Неправильно»	«`InMemoryProjectRepository.RenameAsync` меняет проект без `lock (store.Sync)`. Два параллельных запроса могут потерять одно из изменений — оберни, как в `CreateAsync`»
«Зачем так?»	«Здесь проверка занятости имени идёт после `RenameAsync`, то есть уже после записи. Кажется, её надо перенести выше — или я не вижу, где она ещё?»
«Переделай тесты»	«Нет теста на 404 для несуществующего id. Предлагаю добавить: это ветка, которую легко сломать при рефакторинге»

Мелочи вкуса помечают явно — «nit: можно короче» — чтобы автор понимал, что это не блокирует merge. Автор отвечает на каждый комментарий: исправил (со ссылкой на коммит) или объяснил, почему нет.

8. Инструменты GitHub

Инструмент	Зачем
Issues	Задачи и баги, метки, обсуждение
Projects	Доска Todo → In Progress → Done; закрытые issue уезжают в Done автоматически
Rulesets (защита веток)	Запрет push в `main`, обязательные PR, аппрувы, зелёный CI, запрет force-push
PR template	`.github/pull_request_template.md`: чек-лист в каждом PR
CODEOWNERS	Автоматически назначает ревьюеров по путям
Dependabot	Сам присылает PR с обновлениями NuGet-пакетов, Docker-образов, actions
Releases	Тег `v1.2.3` + заметки о выпуске
Secret scanning	Ловит токены в пушах

> Rulesets на **приватных** репозиториях требуют GitHub Pro. Студентам Pro бесплатен в **GitHub Student Developer Pack**.

Rulesets превращают договорённости в правила, которые нельзя забыть. Договорённость «в `main` только через PR» держится до первого аврала, ruleset — всегда: прямой push отклоняется сервером.

```
remote: error: GH013: Repository rule violations found for refs/heads/main.
remote: - Changes must be made through a pull request.
 ! [remote rejected] main -> main (push declined due to repository rule violations)
```

Правило на `main` в лабе: запрет удаления ветки, обязательный PR с одним аппрувом, запрет force-push. Последнее защищает от переписывания общей истории (п. 5). В теме 5 добавим обязательную проверку CI — тогда красный PR нельзя будет влить даже с аппрувом. Rulesets — более новый механизм, чем классические *branch protection rules*: их можно включать и выключать целиком и накладывать несколько на одну ветку.

Projects — доска поверх issues и PR. Карточки — это сами issue, а не их копии, поэтому статус на доске и в

issue не расходится. Встроенные *Workflows* двигают карточки сами: issue закрыт → **Done**.

Dependabot по файлу `.github/dependabot.yml` раз в неделю проверяет три экосистемы нашего проекта: `nuget` (пакеты в `.csproj`), `docker` (базовые образы в `Dockerfile`) и `github-actions` (версии actions в workflow). На каждое обновление он открывает обычный PR, который проходит тот же CI и то же ревью. Обновления мелкими порциями каждую неделю дешевле, чем раз в год обновлять всё сразу.

9. Релизы и SemVer

`vMAJOR.MINOR.PATCH`:

- **MAJOR** — ломающие изменения API (удалили поле из ответа);
- **MINOR** — новая функциональность без поломок (добавили `/stats`);
- **PATCH** — исправления.

Номер версии — обещание клиентам: по нему видно, можно ли обновиться, не читая код. «Ломающее» определяется не объёмом правки, а тем, перестанет ли работать уже написанный клиент.

Изменение в сервисе задач	Версия
Новый эндпоинт `PUT /api/projects/{id}`	MINOR: старые клиенты его просто не вызывают
Новое необязательное поле в ответе `TaskItem`	MINOR
`GET /api/projects/999` отвечал 500, теперь 404	PATCH
Удалили или переименовали поле `dueDate` в JSON задачи	MAJOR
Новое **обязательное** поле в `CreateTaskRequest`	MAJOR: старые запросы начнут получать 400
`DELETE` стал отвечать 200 вместо 204	MAJOR: клиент мог проверять именно 204

Версии `0.x.y` по SemVer означают начальную разработку, когда API ещё может меняться в любой момент. Поэтому в лабе 4 выпускаем `v0.1.0`, а `v1.0.0` — в лабе 5, когда сервис тестируется и деплоится автоматически.

Тег — неподвижный указатель на коммит (в отличие от ветки). Тег → релиз на GitHub → образ с тегом `1.2.3` в реестре (тема 5). **Release notes** объясняют, что изменилось, тем, кто не читает PR: пользователям API, дежурным, соседним командам. Кнопка *Generate release notes* собирает их из заголовков влитых PR с авторами и ссылкой на полный список изменений. Поэтому хороший заголовок PR — это одновременно строка в журнале изменений.

10. Секреты

- **Никогда не коммитить:** пароли, токены, `.env`, `appsettings.Production.json` с секретами.
- **Секрет утёк — ротировать:** выпустить новый, отозвать старый. Удалённый коммит не спасает: история уже скопирована.
- **Секреты для CI** хранятся в *Settings → Secrets and variables → Actions* (тема 5).

Типичная история. Разработчик пушит в публичный репозиторий ветку, в которой случайно оказался `.env` с токеном облака. Через минуту он замечает, удаляет файл новым коммитом и успокаивается. Но публичные push'и постоянно сканируют боты, и токен в старом коммите уже скопирован. Через некоторое время на аккаунте появляются чужие ресурсы. Удаление файла не помогло, потому что коммит с ним остался в истории и в чужих копиях.

Правильный порядок при утечке:

1. **Отозвать и перевыпустить** секрет — сразу, до всего остального. С этого момента утёкшее значение бесполезно.
2. **Поменять значение** там, где сервис его использует: `.env`, секреты Actions.
3. **Проверить журналы** доступа за время, пока секрет был открыт.
4. **Только потом, если нужно,** вычистить историю (`git filter-repo`) и force-push'нуть. Это сложно, ломает клоны коллег и не удаляет уже сделанные копии, поэтому само по себе ничего не решает.

Секрет считается скомпрометированным с момента push, а не с момента, когда его кто-то нашёл. Для публичных репозиториев GitHub проверяет пуши на известные форматы токенов (*push protection*) и отклоняет такой push, но это страховка, а не замена `.gitignore` и внимательному `git status`.

11. Git как инструмент расследования

История нужна не только чтобы откатиться. Когда после релиза сервис начал отвечать 500, первые вопросы — «что изменилось» и «в каком коммите» — решаются командами Git.

Команда	Вопрос
`git log --oneline v0.1.0..v0.2.0`	Что вошло в релиз между двумя тегами
`git log -S "NameExistsAsync"`	В каких коммитах появилась или исчезла эта строка
`git blame -L 80,105 src/DemoApi/Controllers/ProjectsController.cs`	Кто и в каком коммите последним менял эти строки
`git bisect`	Какой именно коммит сломал поведение
`git revert <хэш>`	Отменить коммит новым коммитом, не переписывая историю

`blame` показывает хэш, а через squash-коммит `(#9)` — PR с обсуждением. Часто там уже написано, почему сделано именно так.

`git bisect` — двоичный поиск по истории. Вы указываете заведомо хороший коммит (например, тег прошлого релиза) и плохой (текущий), Git выгружает середину, вы проверяете и отвечаете `good` или `bad`. На сотню коммитов уходит около семи проверок. Если проверку можно написать скриптом или тестом, `git bisect run` пройдёт всё сам.

```
git bisect start
git bisect bad                  # текущий коммит сломан
git bisect good v0.1.0          # на прошлом релизе всё работало
# ... проверяем, отвечаем git bisect good / bad
git bisect reset                # вернуться туда, где были
```

Типичная история. После релиза переименование проекта начинает иногда отдавать 500. В релиз вошло два десятка PR, по диффу причину не видно. `bisect` между тегами за несколько шагов находит squash-коммит, в котором поменяли SQL-запрос, из PR понятно, что хотели сделать. Сервис чинят `git revert` этого коммита через обычный PR, а нормальное исправление делают отдельной задачей.

`revert`, а не `reset`, для `main`. `git reset --hard` передвигает ветку назад и выбрасывает коммиты — у всех, кто уже их забрал, история разойдётся, а ruleset не даст сделать force-push. `git revert` добавляет новый коммит, отменяющий старый: история честная, изменение проходит через PR и CI, как любое другое. Если что-то пошло не так локально, `git reflog` показывает, куда указывал `HEAD` раньше, — потерянные коммиты почти всегда можно вернуть.

12. Git в Visual Studio

Всё то же доступно без терминала:

- **Git Changes** (*View → Git Changes*): что изменено, индекс, коммит, push, pull, смена ветки;
- **Git Repository**: история и ветки в виде графа;
- **редактор слияния** при конфликтах.

В курсе даём команды: они одинаковы в любой IDE и в CI, а в теме 5 те же действия выполняет пайплайн без всякой IDE.

Типичные ошибки

- **Коммиты прямо в `main`.**
- **PR на 2000 строк:** его не ревьюят, а «аппрувят».
- **Merge при красном CI.**
- **Сообщения «fix», «wip», «asdf».**
- **Разрешение конфликта** выбором «всё моё» без чтения чужого кода.
- **`git add -A` не глядя** — и в репозитории `.env` или `bin/`.
- **Новая ветка от старого `main`** без `git pull` — лишние конфликты.
- **Rebase или force-push общей ветки:** у коллег ломается история.
- **Утёкший секрет «удалили коммитом»** вместо ротации.

Вопросы для самопроверки

1. Что такое ветка с точки зрения Git и почему создать её «дёшево»?
2. Зачем нужен индекс, если можно коммитить сразу всё?
3. Почему изменение старого коммита меняет хэши всех последующих?
4. Когда merge будет fast-forward, а когда появится коммит слияния? Что меняет squash?
5. Две ветки поменяли разные строки `ProjectsController.cs`. Будет ли конфликт? А если одну и ту же?
6. Чем `git revert` лучше `git reset` для `main`?
7. Какие правила вы поставите на `main` и почему?
8. Какая версия выйдет после удаления поля из JSON задачи? После добавления нового эндпоинта?
9. Утёк токен в публичный репозиторий. Порядок действий?

Практика: лаба 4

Выложите репозиторий сервиса задач в GitHub и защитите `main`. Через issue и PR с ревью добавите README и новую функцию — переименование проекта — в обоих хранилищах. Разрешите конфликт с партнёром и выпустите релиз `v0.1.0`.