Неделя 10. Лекция 5. CI/CD: GitHub Actions и автоматический деплой

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

7. Стратегии деплоя

Стратегия	Как	Простой	Ресурсы
Recreate	Остановить старую версию, запустить новую	Секунды, пока стартует новая	Один экземпляр
Rolling	Заменять экземпляры по одному	Нет, если экземпляров несколько	Несколько экземпляров за балансировщиком
Blue-green	Поднять новую версию рядом, переключить трафик целиком	Нет; откат — переключение назад	Два полных окружения
Canary	Отдать новой версии малую долю трафика, по метрикам расширять	Нет	Балансировщик с весами и мониторинг

У нас **recreate**: одна машина, один экземпляр, порт 8080 занят одним контейнером. Несколько секунд простоя для учебного сервиса приемлемы, а сложность других стратегий — нет. Rolling, blue-green и canary дают оркестраторы (например, Kubernetes) и балансировщики. У всех трёх есть общее условие: какое-то время старая и новая версии работают с одной БД одновременно. Поэтому изменения схемы делают обратно совместимыми — сначала добавляют колонку, и только через релиз удаляют старую.

8. Откат

Плохой релиз? **Откат = деплой предыдущего `sha-` тега.** Пересобирать ничего не нужно: образ уже есть в реестре и в локальном кэше Docker.

```
        $env:API_IMAGE = "ghcr.io/<you>/demo-api"; $env:API_TAG = "sha-<предыдущий>"
        docker compose up -d --no-build --wait
```

Откат быстрый, потому что пропускает всё медленное: сборку, тесты, загрузку образа. Он возвращает код, но не данные: если новая версия успела изменить схему или записи в БД, старый образ увидит их такими, какие они есть. Ещё одна тонкость: ручной откат живёт до следующего push в `main`, и новый запуск пайплайна задеплоит свою версию поверх. Поэтому после отката исправление идёт обычным путём, через PR (лаба 8).

Правило эксплуатации: **сначала восстановить сервис, потом разбираться** (лаба 8).

9. Секреты, права, окружения

- **`GITHUB_TOKEN`** создаётся на каждый запуск, действует только на этот репозиторий и истекает после job. На уровне workflow задано `contents: read`. Блок `permissions` у job не дополняет, а **заменяет** общий, поэтому в `image` и `deploy` `contents: read` повторён рядом с `packages: write` и `packages: read`.
- **Секреты** (`DB_PASSWORD`) — в *Settings → Secrets and variables → Actions*, в workflow — `${{ secrets.DB_PASSWORD }}`. В логах GitHub их маскирует, но только точное значение: закодированный или разрезанный секрет в лог утечёт. Запускам из PR из форков секреты не передаются, а `GITHUB_TOKEN` там только на чтение.
- **Variables** (`${{ vars.X }}`) — несекретные настройки: порт, имя окружения. Их видно в интерфейсе и в логах.
- **Environments** (*Settings → Environments*) дают ручное одобрение деплоя (*Required reviewers*), секреты окружения и ограничение веток. Job с `environment:` ждёт одобрения в статусе *Waiting*. Для приватных репозиториев часть функций только на платных планах.
- **`concurrency`** + `cancel-in-progress` — новый push отменяет устаревший запуск. Группа `${{ github.workflow }}-${{ github.ref }}` своя у каждой ветки и каждого PR. Оборотная сторона: второй push в `main` может прервать `deploy` на середине. Для нас это терпимо, следующий запуск всё равно задеплоит, но в production деплою обычно дают отдельную группу без отмены.

10. Обязательные проверки и Dependabot

Ruleset из лекции 4 запрещает прямой push в `main`. **Require status checks to pass** добавляет второе условие: merge возможен, только если job `test` зелёная. Имя проверки совпадает с именем job. Источник GitHub Actions выбирают, чтобы засчитывался результат именно пайплайна, а не статус, который прислала бы другая интеграция. Опция *Require branches to be up to date* заставляет перед merge подтянуть свежий `main` и прогнать тесты заново: два PR, зелёных по отдельности, вместе могут сломать сборку.

Dependabot присылает PR с обновлениями NuGet-пакетов, базовых образов и самих actions — например, новой major-версии `actions/checkout`. Эти PR проходят тот же `test`, что и ваши, и мержатся по тем же правилам. Обновление зависимости — такое же изменение кода: зелёный CI на PR от Dependabot — главный аргумент, что его можно принять. Секреты Actions в такие запуски не передаются, но нашему `test` они и не нужны.

11. Как понять, что пайплайн помогает: метрики DORA

Исследовательская программа DORA выделила четыре метрики поставки, которые одновременно говорят о скорости и о стабильности:

Метрика	Что измеряет	Где взять у нас
Частота деплоя	Как часто изменения попадают к пользователям	Число успешных `deploy` в *Actions*
Lead time for changes	Время от коммита до работы в production	От merge до зелёного smoke test
Change failure rate	Доля деплоев, после которых нужен откат или hotfix	Откаты и срочные PR после релизов
Время восстановления (MTTR)	Сколько сервис живёт в сбое	От алерта до отката (лаба 8)

Первые две метрики улучшает CI/CD: мелкие изменения выходят часто и быстро. Две последние требуют мониторинга: чтобы посчитать сбои и время восстановления, сбой сначала нужно заметить. Это тема 7.

Типичные ошибки

- **`permissions` не заданы** или `write-all`.
- **Секреты в YAML или в выводе `echo`.**
- **Деплой `latest`:** неизвестно, что запущено, откат невозможен.
- **Деплой без `--wait` и smoke test:** «зелёный» пайплайн с лежащим сервисом.
- **Интеграционные тесты требуют БД, а в CI её нет.** Решение — `services:`.
- **Service container без health check:** тесты падают «через раз», пока БД стартует.
- **Имя образа с заглавными буквами** (`ghcr.io/UserName/...`): push падает.
- **Сборка образа на машине деплоя** (`up` без `--no-build`): в production работает не то, что проверили тесты.
- **Runner установлен как служба Windows:** job падает, не находя Docker Desktop.
- **Self-hosted runner на публичном репозитории без одобрения внешних PR.**
- **Не включена обязательная проверка `test`:** красный PR мержится одной кнопкой.

Вопросы для самопроверки

1. Почему на PR не собирается и не пушится образ?
2. Откуда в CI берётся PostgreSQL для тестов и как тесты узнают, что нужно работать с ним?
3. Зачем деплоить по `sha-` тегу, если есть `latest`?
4. Как GitHub достучится до self-hosted runner за NAT?
5. Как откатиться за минуту без пересборки?
6. Как имя образа и тег попадают из job `image` в job `deploy`? Почему нельзя просто записать их в файл в `image`?
7. Чем Continuous Delivery отличается от Continuous Deployment? Что из них делает пайплайн на push в `main`, а что — на тег?
8. Что произойдёт, если повторно запустить `deploy` для уже развёрнутого тега?

Практика: лаба 5

Подключите пайплайн к репозиторию сервиса задач: тесты на каждый PR и запрет merge при красных тестах, образ в GHCR, автодеплой на ваш Docker Desktop через self-hosted runner и откат на предыдущую версию. Проведите изменение через весь путь — от ветки до нового `version` в `GET /` — и выпустите релиз `v1.0.0`.