Неделя 9. Лекция 5. CI/CD: GitHub Actions и автоматический деплой

Часть 1 из 2.

После лекции: вы понимаете, зачем нужны CI и CD и чем Continuous Delivery отличается от Continuous Deployment, как GitHub Actions исполняет workflow, и можете объяснить каждую строку пайплайна сервиса задач: тесты с PostgreSQL → образ в реестр → деплой на Docker Desktop → проверка → откат. **Инструменты:** GitHub Actions, GitHub Container Registry (GHCR), self-hosted runner, Docker Desktop, PowerShell 7.

1. Зачем CI/CD

Сейчас, чтобы выкатить новую версию сервиса, вы вручную: запускаете тесты (если не забыли), собираете образ, останавливаете контейнер, запускаете новый. Каждый ручной шаг — шанс ошибиться, а «забыл запустить тесты» — классика. К тому же процесс живёт в голове одного человека: в отпуске он — стоит и выкладка.

Сломанный `main` — состояние, когда главная ветка не собирается или в ней красные тесты. От этого страдают все: каждый, кто создал ветку от `main`, получает чужую ошибку, и уже непонятно, какой из последних коммитов виноват. Цена ошибки растёт со временем. Через минуту после push автор помнит, что менял, и исправляет за минуту. Через неделю ошибку ищут по истории из десятков коммитов. У пользователей к поиску добавляются испорченные данные, откат и разбор инцидента.

Практика	Что автоматизировано	Кто решает о выкладке
CI (Continuous Integration)	Сборка и тесты каждого изменения	—
Continuous Delivery	CI + готовый к выкладке артефакт (у нас — образ)	Человек, нажатием кнопки
Continuous Deployment	Всё, включая выкладку после merge в `main`	Зелёный пайплайн

CI — прежде всего практика команды, а не сервер: маленькие изменения часто вливаются в `main`, и каждое проверяется автоматически. Инструмент лишь делает проверку дешёвой. Continuous Deployment требует доверия к тестам и быстрого отката: без них автоматическая выкладка просто быстрее доставляет ошибки пользователям. У нас push в `main` деплоится автоматически, а тег релиза только публикует образ — это уже Delivery.

История из практики. Разработчик добавил поле в задачу, прогнал тесты на хранилище в памяти, а в SQL-репозитории поле забыл. Локально всё зелёное, ревьюер тоже смотрел на логику, а не на SQL. Ошибка всплыла после выкладки, когда первые пользователи открыли задачи. Наш CI гоняет тесты на настоящем PostgreSQL, и такое изменение останется красным ещё в PR.

Результат: merge в `main` → через пару минут новая версия работает, а сломанный код до `main` просто не доходит.

2. Как устроен GitHub Actions

```yaml
# .github/workflows/ci-cd.yml
name: ci-cd
on:                                  # события-триггеры
  pull_request: { branches: [main] }
  push: { branches: [main], tags: ["v*.*.*"] }

permissions:
  contents: read                     # минимальные права GITHUB_TOKEN

jobs:
  test:                              # job — набор шагов на одной машине
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v7    # готовый action: скачать код
      - run: dotnet test             # shell-команда
  image:
    needs: test                      # после test
    if: github.event_name == 'push'  # только на push
```

Понятие	Суть
Workflow	YAML-файл в `.github/workflows/`
Event	Что запускает: `push`, `pull_request`, `workflow_dispatch` (кнопка), `schedule`
Job	Набор шагов на отдельной машине (runner). Jobs идут параллельно, если нет `needs`
Step	`uses:` — готовый action, `run:` — команда
Runner	Машина, где выполняется job: GitHub-hosted (`ubuntu-latest`) или **self-hosted** (ваша)
Контексты	`${{ github.sha }}`, `${{ secrets.X }}`, `${{ needs.image.outputs.tag }}`
`GITHUB_TOKEN`	Временный токен запуска; права задаются в `permissions`
Service containers	Контейнеры рядом с job — например, PostgreSQL для интеграционных тестов

Что происходит после push. GitHub получает событие и ищет в `.github/workflows/` файлы, у которых `on:` совпадает с ним. Для `pull_request` берётся версия workflow из самого PR, поэтому изменения пайплайна проверяются до merge. Из `needs` строится граф jobs, и каждая job встаёт в очередь на runner с подходящей меткой `runs-on`. Runner выполняет шаги по порядку в одной рабочей папке. Упал шаг — следующие пропускаются, job красная, а зависящие от неё jobs не запускаются вовсе.

Jobs изолированы. Каждая job — отдельная машина: файлы, переменные и собранные бинарники из `test` в `image` не видны. Данные между jobs передают тремя способами. **Outputs** — короткие строки, как имя и тег образа. **Артефакты** (`actions/upload-artifact` и `download-artifact`) — файлы: отчёты тестов, архивы. **Внешнее хранилище** — у нас реестр: `image` кладёт туда образ, `deploy` забирает его по тегу.

Каждый запуск на GitHub-hosted runner стартует на **чистой машине**: на ней нет ваших файлов и настроек, только то, что описано в workflow. Поэтому «у меня работает» тут не пройдёт.

	GitHub-hosted	Self-hosted
Где	Виртуальная машина в облаке GitHub	Ваш ПК или сервер
Состояние	Новая машина на каждую job, после — удаляется	Сохраняется между запусками: рабочая папка, образы Docker
Сеть	Интернет, но не ваша локальная сеть	Всё, что видит ваша машина, включая Docker Desktop
Кто обслуживает	GitHub	Вы: обновления, диск, безопасность

3. Пайплайн сервиса задач

```
flowchart LR
  PR["Pull Request"] --> T1["test<br/>dotnet test + Postgres"]
  M["push в main"] --> T2["test"] --> I["image<br/>build → GHCR<br/>sha-abc1234, latest"] --> D["deploy<br/>self-hosted runner<br/>compose pull / up --wait"] --> S["smoke<br/>/health/ready"]
  TAG["тег v1.2.0"] --> T3["test"] --> I2["image<br/>1.2.0"]
```

Событие	test	image	deploy
Pull Request	да	—	—
push в `main`	да	да: `sha-xxxxxxx`, `latest`	да
тег `v1.2.0`	да	да: `1.2.0`, `sha-xxxxxxx`	—

Файл пайплайна: `kit/.github/workflows/ci-cd.yml` (лаба 5).

Почему на PR нет образа. Код в PR ещё не прошёл ревью, а публикация требует права на запись в реестр. Выдавать его непроверенному коду, тем более из чужого форка, незачем. К тому же реестр быстро засорился бы образами из каждого коммита каждой ветки.

Почему тег не деплоится. Тег ставят на коммит, который уже прошёл через `main` и уже развёрнут. Релиз фиксирует версию под понятным именем `1.2.0`, а не выкладывает что-то новое.

4. Job `test`: тесты с настоящей БД

```yaml
  test:
    runs-on: ubuntu-latest
    services:
      postgres:
        image: postgres:18-alpine
        env:
          POSTGRES_DB: demo
          POSTGRES_USER: demo
          POSTGRES_PASSWORD: demo
        ports:
          - 5432:5432
        options: >-
          --health-cmd "pg_isready -U demo -d demo"
          --health-interval 5s --health-timeout 3s --health-retries 10
    env:
      Storage: Postgres
      ConnectionStrings__Db: Host=localhost;Port=5432;Database=demo;Username=demo;Password=demo
    steps:
      - uses: actions/checkout@v7
      - uses: actions/setup-dotnet@v6
        with:
          dotnet-version: 10.0.x
      - run: dotnet restore
      - run: dotnet build --no-restore -c Release
      - run: dotnet test --no-build -c Release
```

- **`services.postgres`.** До первого шага runner запускает контейнер с тем же образом `postgres:18-alpine`, что и в `compose.yaml`. Тесты идут на той же версии БД, что работает после деплоя. Service containers на GitHub-hosted runner доступны только на Linux, поэтому `ubuntu-latest`.
- **`ports`.** Сама job выполняется прямо на виртуальной машине, а не в контейнере. Поэтому к БД обращаемся по `localhost` и опубликованному порту.
- **`options` с `--health-cmd`.** Runner не начнёт шаги, пока PostgreSQL не станет `healthy`. Без этого первые тесты стучались бы в ещё не готовую БД и падали «через раз» — самый неприятный вид красного теста.
- **`env`.** Слои конфигурации из лекции 1: переменные окружения перекрывают `appsettings*.json`. `WebApplicationFactory` поднимает `Program` с этой же конфигурацией, и тесты переключаются на PostgreSQL без правки кода.
- **`restore` → `build` → `test` отдельными шагами.** В логе сразу видно, где упало: NuGet и сеть, компиляция или тесты. `--no-restore` и `--no-build` не повторяют сделанную работу, а `-c Release` проверяет ту же конфигурацию, что публикует Dockerfile.

Упавший тест в логе шага выглядит примерно так:

```
Failed DemoApi.Tests.Integration.ProjectsApiTests.Create_DuplicateName_ReturnsConflict [41 ms]
  Error Message:
   Assert.Equal() Failure: Values differ
Expected: OK
Actual:   Conflict
```

Имя теста, ожидание и факт — этого достаточно, чтобы понять причину, не запуская ничего локально.

Кэширование. Сейчас `dotnet restore` каждый раз скачивает пакеты заново. `actions/setup-dotnet` умеет кэшировать их (параметр `cache: true`), но ему нужен lock-файл `packages.lock.json`. Для нашего размера проекта выигрыш — секунды, в больших решениях — минуты на каждом запуске.

> **Демонстрация для преподавателя.** Откройте PR и вкладку *Checks*: работает только `test`, `image` и `deploy` помечены *skipped*. Раскройте шаг *Initialize containers* — видно, как runner скачивает `postgres:18-alpine` и ждёт health check. Затем сломайте один `Assert` в тесте, запушьте и покажите, где в логе имя теста и строки `Expected/Actual`.

5. Job `image`: образ в реестр

```yaml
  image:
    needs: test
    if: github.event_name == 'push'
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write
    outputs:
      image: ${{ steps.vars.outputs.image }}
      tag: ${{ steps.vars.outputs.tag }}
    steps:
      - id: vars
        run: |
          echo "image=ghcr.io/${GITHUB_REPOSITORY,,}" >> "$GITHUB_OUTPUT"
          echo "tag=sha-${GITHUB_SHA::7}" >> "$GITHUB_OUTPUT"
```

Шаг `vars` работает в bash (оболочка по умолчанию на Linux). `${GITHUB_REPOSITORY,,}` переводит `owner/repo` в нижний регистр, `${GITHUB_SHA::7}` берёт первые 7 символов sha. Строки `ключ=значение`, дописанные в файл `$GITHUB_OUTPUT`, становятся outputs шага, а блок `outputs:` выносит их на уровень job — оттуда их прочитает `deploy`.

- **Реестр GHCR (GitHub Container Registry):** `ghcr.io/<owner>/<repo>:<tag>`, **только в нижнем регистре**. Push идёт через `GITHUB_TOKEN` с правом `packages: write`. Пакет приватный по умолчанию, виден на странице репозитория → *Packages*.
- **`docker/setup-buildx-action@v4`** готовит сборщик BuildKit: без него не работает экспорт кэша в Actions.
- **`docker/login-action@v4`** входит в `ghcr.io` от имени `github.actor` с `GITHUB_TOKEN` — отдельный токен доступа создавать не нужно.
- **`docker/metadata-action@v6`** по событию вычисляет список тегов и OCI-метки: ссылку на репозиторий, sha коммита. По меткам со страницы пакета видно, из какого кода он собран.
- **`docker/build-push-action@v7`** собирает тот же ваш Dockerfile и сразу пушит. `build-args: APP_VERSION=sha-abc1234` попадает в `ARG APP_VERSION`, оттуда в переменную окружения, и её показывает `GET /`.
- **Кэш слоёв** `cache-from: type=gha`, `cache-to: type=gha,mode=max` хранит слои в кэше GitHub Actions. `mode=max` сохраняет и промежуточный этап `build`. Если `.csproj` не менялся, слой с `dotnet restore` берётся из кэша даже на новой машине. Кэш ограничен по объёму, старые записи GitHub вытесняет сам.

Стратегия тегов:

Тег	Для чего
`sha-abc1234`	**Деплой.** Неизменяемый: точно известно, какой коммит запущен
`1.2.0`	Релиз
`latest`	Удобство, **не для деплоя**

Почему деплоим по sha, а не `latest`. `latest` — подвижный ярлык: сегодня он указывает на один образ, завтра на другой. Если на сервере `latest`, на вопрос «какой код сейчас работает» ответа нет, а для отката нечего указать — прошлый `latest` уже перезаписан. Тег `sha-abc1234` связывает контейнер, коммит и запуск пайплайна в одну цепочку. Технически перезаписать можно и его, но наш пайплайн пишет его один раз на коммит, и на практике он неизменяем.

```mermaid
flowchart LR
  T["test<br/>VM GitHub"] -->|"needs: только статус"| I["image<br/>VM GitHub"]
  I -->|"docker push<br/>sha-abc1234"| R[("GHCR")]
  I -->|"outputs: image, tag"| D["deploy<br/>ваш ПК"]
  R -->|"docker compose pull"| D
```

6. Job `deploy`: self-hosted runner и Docker Desktop

```yaml
  deploy:
    needs: image
    if: github.ref == 'refs/heads/main'
    runs-on: self-hosted
    permissions:
      contents: read
      packages: read
    defaults:
      run:
        shell: pwsh          # на машине с runner нужен PowerShell 7
    env:
      API_IMAGE: ${{ needs.image.outputs.image }}
      API_TAG: ${{ needs.image.outputs.tag }}
      DB_PASSWORD: ${{ secrets.DB_PASSWORD }}
```

Машина GitHub не видит ваш компьютер. Поэтому деплой выполняет **self-hosted runner** — агент на вашем ПК:

- Агент **сам опрашивает GitHub** (только исходящие соединения по HTTPS). Белый IP и открытые порты не нужны.
- Job с `runs-on: self-hosted` выполняется на вашей машине и управляет вашим Docker Desktop.
- **Запускайте `run.cmd` в своей сессии.** Runner как служба Windows работает от системной учётки без доступа к Docker Desktop.
- **Шаги деплоя выполняются в PowerShell 7** (`shell: pwsh`).

Откуда берётся образ. `env` связывает пайплайн с вашим `compose.yaml`: там `image: ${API_IMAGE:-demo-api}:${API_TAG:-local}`. Локально переменных нет, и Compose собирает `demo-api:local`. В деплое `API_IMAGE` и `API_TAG` приходят из outputs job `image`, и сервис запускается из образа реестра. `DB_PASSWORD` берётся из секрета и должен совпадать с паролем, с которым когда-то инициализирован том `demo_pgdata`. `actions/checkout` нужен ради `compose.yaml` и папки `db/`, а `docker/login-action` здесь хватает права `packages: read`.

```yaml
    - run: docker compose pull api                                  # скачать образ sha-... из GHCR
    - run: docker compose up -d --no-build --wait --wait-timeout 90    # пересоздать контейнер и дождаться healthy
    - run: docker compose exec -T api curl -fsS http://localhost:8080/health/ready   # smoke test
```

- **`--no-build`:** на машине деплоя ничего не собирается. Работает ровно тот образ, который прошёл тесты и лежит в реестре.
- **Пересоздаётся только `api`:** у него изменился образ. Контейнер `db` и его том Compose не трогает.
- **`--wait`:** job упадёт, если новый контейнер не стал `healthy` за 90 секунд. Статус даёт `HEALTHCHECK` из Dockerfile, то есть `/health/live`: процесс жив.
- **Smoke test** — `/health/ready` проверяет ещё и доступность БД. `-T` отключает псевдотерминал, которого у runner нет, а `curl -f` превращает ответ 503 в ненулевой код выхода и красный шаг.

Идемпотентность. Повторный деплой того же тега ничего не ломает: Compose сравнивает желаемое состояние с текущим и, если образ тот же, оставляет контейнеры как есть. Поэтому *Re-run all jobs* безопасен и годится для возврата на актуальную версию после ручного отката. В логе шага *Deploy* это выглядит примерно так:

```
Container demo-db-1  Running
Container demo-api-1  Recreate
Container demo-api-1  Recreated
Container demo-api-1  Started
Container demo-api-1  Waiting
Container demo-api-1  Healthy
```

При повторном запуске того же тега строк `Recreate` не будет — оба контейнера останутся `Running`.

История из практики. Пайплайн заканчивался на `docker compose up -d`. Команда возвращает управление сразу после старта контейнера, а сервис падал через пару секунд из-за неверной строки подключения. Пайплайн зелёный, в чате «задеплоено», пользователи получают ошибки, а заметили это по жалобам. `--wait` и smoke test делают такой деплой красным, и его видно сразу.

Риски self-hosted runner. Runner выполняет любой код, который попал в workflow, с правами вашей учётной записи. В публичном репозитории любой может открыть PR из форка, прописать в нём `runs-on: self-hosted` и выполнить свои команды на вашем ПК. GitHub прямо рекомендует self-hosted runner только для приватных репозиториев. В публичном обязательно требуйте одобрения запусков от внешних контрибьюторов (лаба 4, п. 1). Второй риск — состояние между запусками: образы, файлы в рабочей папке,

логин в реестр остаются на машине. Для нас это плюс (предыдущий образ уже в кэше для отката), но чистоты GitHub-hosted здесь нет.

```
sequenceDiagram
  participant GH as GitHub
  participant R as Runner на вашем ПК
  participant REG as GHCR
  participant DD as Docker Desktop
  R->>GH: есть задания? (исходящий HTTPS)
  GH-->>R: job deploy, тег sha-abc1234
  R->>REG: docker compose pull api
  REG-->>R: образ sha-abc1234
  R->>DD: compose up -d --no-build --wait
  DD-->>R: api healthy
  R->>DD: exec curl /health/ready
  DD-->>R: 200
  R->>GH: job успешна, лог шагов
```

> **Демонстрация для преподавателя.** Поставьте рядом окно `run.cmd` и страницу *Actions*. После merge в окне runner появляется `Running job: deploy`, затем `Job deploy completed with result: Succeeded`. Выполните `irm http://localhost:8080/` и сравните `version` с sha коммита в `main`. Затем нажмите *Re-run jobs* для `deploy` и покажите, что контейнер `api` не пересоздан: в `docker compose ps` колонка `STATUS` показывает прежний аптайм.