Лекция 15: CI/CD. Строим "заводской конвейер" для нашего приложения

Введение: Боль ручного релиза

Здравствуйте! На предыдущих лекциях мы создали сборочный чертеж (docker-compose.yml) для нашего "автомобиля" (приложения). Но пока что сборка все еще происходит вручную в "гараже" — на нашей локальной машине.

Представьте, как выглядит процесс выпуска новой версии в компании, где нет автоматизации. В пятницу вечером "релизный инженер" Вася заходит на сервер, вручную выполняет git pull, npm install, npm test, docker build, docker-compose up... Этот процесс полон рисков: Вася может перепутать ветку, забыть выполнить шаг, опечататься в команде. Каждый релиз — это стресс, риск и потенциальная работа на выходных. Команды боятся релизов, и новые фичи доходят до пользователей месяцами.

CI/CD (Continuous Integration / Continuous Delivery) — это философия и набор практик, которые превращают этот рискованный ручной ритуал в скучный, предсказуемый и полностью автоматический **заводской конвейер**.

Аналогия:

- **Ручной релиз** — это когда один мастер **вручную** собирает автомобиль.
- **CI/CD Pipeline** — это **заводской конвейер Генри Форда**. Код (шасси) движется по ленте, и на каждом этапе роботы (скрипты) автоматически выполняют свою операцию: установка зависимостей, запуск тестов, сборка Docker-образа.

1. Расшифровываем акронимы: CI, CD, CD

CI — Continuous Integration (Непрерывная Интеграция)
Это практика, при которой разработчики часто (несколько раз в день) вливают свои изменения в общую основную ветку репозитория. **Главное:** после **каждого** такого вливания автоматически запускается процесс **сборки и тестирования** проекта.

- **Цель CI:** Как можно раньше обнаружить ошибки интеграции. Если новый код "сломал" сборку или тесты, команда узнает об этом в течение минут, а не перед релизом. Это "робот-контролер", который проверяет каждую новую деталь на конвейере.

CD — Continuous Delivery (Непрерывная Доставка)
Это логическое продолжение CI. Если код успешно прошел все этапы CI (сборку и все тесты), он автоматически "упаковывается" в готовый к выпуску **артефакт** (в нашем случае — Docker-образ) и может быть развернут на тестовый стенд (staging).

- **Ключевой момент:** Финальное развертывание на "боевой" сервер (production) требует **ручного подтверждения** от человека (например, нажатия кнопки). Мы уверены, что наш "автомобиль" готов сойти с конвейера, но решение, когда именно это сделать, принимает менеджер.

CD — Continuous Deployment (Непрерывное Развертывание)
Это высший пилотаж. Каждое изменение, успешно прошедшее все тесты, автоматически, **без**

какого-либо ручного вмешательства, развертывается на продакшн. Этот подход требует очень высокой культуры тестирования и мониторинга и используется IT-гигантами, которые могут делать сотни релизов в день.

Для большинства компаний целью является **Continuous Delivery**.

2. Анатомия конвейера (Pipeline)

Пайплайн — это и есть наш автоматизированный рабочий процесс, описанный в виде кода. Он состоит из этапов и задач. Современный подход — **Pipeline-as-Code**, когда вся логика пайплайна описывается в YAML-файле, который хранится в репозитории вместе с кодом.

Популярные CI/CD инструменты:

- **Jenkins:** "Дедушка" CI/CD, очень мощный, но сложный в настройке.
- **GitLab CI/CD:** Мощный инструмент, глубоко интегрированный в GitLab.
- **GitHub Actions:** Относительно новый, но невероятно популярный и удобный, встроенный прямо в GitHub.

В рамках нашего курса мы будем использовать **GitHub Actions**.

3. Практика: Создаем CI-пайплайн для нашего API

Давайте создадим "конвейер" для нашего приложения.

Цель: Создать workflow, который будет запускаться при каждом push в ветку main и при создании Pull Request.
Этот workflow должен:

1. Подготовить окружение (скачать код, установить Node.js).
2. Установить зависимости.
3. **Запустить наши тесты (npm test)**.
4. (Если тесты прошли) **Собрать Docker-образ**.
5. **Загрузить (push) Docker-образ** в реестр контейнеров (GitHub Packages).

Шаг 1: Создаем файл .github/workflows/ci.yml
Вся магия GitHub Actions живет в этой папке в корне нашего проекта.

Шаг 2: Описываем пайплайн
YAML-файл GitHub Actions имеет простую структуру: событие (on) -> задание (job) -> шаги (steps).

```
# .github/workflows/ci.yml
name: Node.js CI/CD

# 1. Триггеры: запускать при пуше или PR в ветку main
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

# 2. Задания
jobs:
```

```
# Название нашего задания
build-test-and-push:
  # 3. На какой виртуальной машине запускать
  runs-on: ubuntu-latest

  # 4. Шаги, которые нужно выполнить
  steps:
    # Шаг 1: Скачать код репозитория на виртуальную машину
    - name: Checkout repository
      uses: actions/checkout@v3

    # Шаг 2: Настроить Node.js и кэширование npm
    - name: Setup Node.js
      uses: actions/setup-node@v3
      with:
        node-version: 18
        cache: 'npm'

    # Шаг 3: Установить зависимости (быстро, благодаря кэшу)
    - name: Install dependencies
      run: npm ci

    # Шаг 4: Запустить наши тесты
    - name: Run tests
      run: npm test

    # Шаг 5: Войти в реестр контейнеров GitHub
    # Выполняется только при пуше в main, а не на PR
    - name: Log in to GitHub Container Registry
      if: github.event_name == 'push' && github.ref == 'refs/heads/main'
      uses: docker/login-action@v2
      with:
        registry: ghcr.io
        username: ${{ github.actor }}
        password: ${{ secrets.GITHUB_TOKEN }}

    # Шаг 6: Собрать и загрузить Docker-образ
    - name: Build and push Docker image
      if: github.event_name == 'push' && github.ref == 'refs/heads/main'
      uses: docker/build-push-action@v4
      with:
        context: .
        push: true
        # Тегируем образ уникальным хэшем коммита
        tags: ghcr.io/${{ github.repository }}:${{ github.sha }}
```

Разбор ключевых моментов:

- uses: actions/...: Мы используем готовые, переиспользуемые "**экшены**" из Marketplace для типовых задач (скачивание кода, настройка Node.js, логин в Docker).
- run: npm ci: Мы используем npm ci вместо npm install, так как эта команда гарантирует установку точных версий из package-lock.json, обеспечивая воспроизводимость сборки.
- if: github.event_name == 'push' ...: Мы не хотим собирать и публиковать Docker-образ для каждого коммита в Pull Request. Мы делаем это только тогда, когда код вливается в основную ветку main.

- ${{ secrets.GITHUB_TOKEN }}: GitHub Actions автоматически предоставляет временный токен для безопасной аутентификации в своем реестре.
- tags: ...:${{ github.sha }}: Мы тегируем каждый образ уникальным **хэшем коммита**. Это обеспечивает **полную прослеживаемость**: мы всегда знаем, какая именно версия кода "зашита" в конкретном Docker-образе.

4. От CI к CD: Автоматизируем доставку

Наш CI-пайплайн теперь автоматически создает готовый к развертыванию артефакт. Следующий шаг — доставка его на сервер.

Мы можем добавить в наш workflow еще одно задание (job), которое будет выполняться **после** успешной сборки.

```
# ... (продолжение ci.yml)
  # Новое задание для деплоя
  deploy-to-staging:
    # Запускать только после успешного завершения build-test-and-push
    needs: build-test-and-push
    runs-on: ubuntu-latest

    steps:
      - name: Deploy to Staging Server
        uses: appleboy/ssh-action@master
        with:
          host: ${{ secrets.STAGING_HOST }}
          username: ${{ secrets.STAGING_USER }}
          key: ${{ secrets.STAGING_SSH_KEY }}
          script: |
            # Команды, которые будут выполнены на удаленном сервере
            docker pull ghcr.io/${{ github.repository }}:${{ github.sha }}
            docker stop my-app || true
            docker rm my-app || true
            docker run -d --name my-app -p 3000:3000 ghcr.io/$
{{ github.repository }}:${{ github.sha }}
```

Как это работает:

1. Мы заранее добавляем IP-адрес и SSH-ключ от нашего тестового сервера в **GitHub Secrets** — безопасное хранилище для секретных данных.
2. Новое задание deploy-to-staging использует специальный экшен, который подключается к нашему серверу по SSH.
3. На сервере выполняются команды: скачать (pull) новый образ из реестра и перезапустить контейнер с этим новым образом.

Для развертывания на продакшн можно добавить еще одно задание, которое будет запускаться только после **ручного подтверждения** в интерфейсе GitHub, реализуя таким образом полноценный **Continuous Delivery**.

Заключение

Сегодня мы построили "заводской конвейер" для нашего приложения. Мы превратили ручной, рискованный процесс релиза в полностью автоматизированный, надежный и быстрый CI/CD пайплайн.

Теперь любой push в наш репозиторий запускает каскад проверок, гарантируя, что мы не выкатим сломанный код. Результатом работы конвейера является готовый к развертыванию Docker-образ, который можно доставить на любой сервер одной командой.

Мы почти у цели. У нас есть автоматизированный конвейер. Но как управлять нашим приложением, когда у нас не один контейнер, а десятки? Как обеспечить его отказоустойчивость и масштабирование? На следующей лекции мы познакомимся с миром **оркестрации** и его королем — **Kubernetes**.