

Дисципліна

«Бекенд-розробка, сервіси та мікросервіси»

КОНСПЕКТ ЛЕКЦІЇ. ТЕМА 16

CI/CD: автоматизація збірки, тестування та розгортання

Спеціальність: Комп'ютерна інженерія, 3 курс

Обсяг: 1 год 20 хв (лекція 16 з 18)

План лекції

1. Що таке CI/CD і яку проблему він вирішує
2. Continuous Integration: автоматизація тестування
3. Етапи типового CI/CD пайплайну
4. GitHub Actions: практичний workflow для PHP
5. Continuous Delivery vs Continuous Deployment
6. Стратегії розгортання: rolling update, blue-green
7. Повний приклад: від коміту до production

Мета лекції: зрозуміти, як автоматизувати шлях коду від коміту розробника до працюючого production-застосунку, і навчитися описувати найпростіший CI/CD пайплайн для PHP-проекту засобами GitHub Actions.

1. Що таке CI/CD і яку проблему він вирішує

Уявімо типову ситуацію без автоматизації: розробник вносить зміну в код, вручну запускає тести (тема 11) на своїй машині, вручну підключається до сервера, вручну копіює файли, вручну перезапускає застосунок. Кожен крок — можливість для людської помилки: забути запустити тести, скопіювати не той файл, перезапустити не той сервіс.

CI/CD (Continuous Integration / Continuous Delivery, або Deployment) — практика автоматизації цього шляху: спеціальна система сама, без участі людини, забирає новий код, збирає його, запускає тести, і за потреби розгортає на сервері — щоразу однаково, без пропущених кроків.

Абревіатура	Розшифровка
CI	Continuous Integration — безперервна інтеграція (автоматична збірка й тестування)
CD (Delivery)	Continuous Delivery — безперервна доставка (готовність до розгортання, останній крок — вручну)
CD (Deployment)	Continuous Deployment — безперервне розгортання (автоматично, без ручного підтвердження)

"CD" — навмисно неоднозначна абревіатура: у різних командах вона означає або Continuous Delivery (код завжди готовий до розгортання, але фактичне розгортання на production підтверджує людина), або Continuous Deployment (кожна успішна зміна автоматично потрапляє на production без підтвердження). Різниця розглядається детальніше в розділі 5.

2. Continuous Integration: автоматизація тестування

Continuous Integration (CI) — практика, за якої розробники часто (кілька разів на день) інтегрують свої зміни у спільну гілку коду, і кожна така інтеграція автоматично перевіряється: код збирається, запускаються unit- і функціональні тести (тема 11), перевіряється стиль коду.

Головна мета CI — виявити проблему (код не збирається, тест провалився, зламано сумісність з чужими змінами) якомога швидше, одразу після внесення зміни, а не через тиждень, коли причину знайти вже значно складніше.

```
Розробник → git push → CI-система автоматично:  
1. Завантажує залежності (composer install)  
2. Запускає unit-тести (тема 11, PHPUnit)  
3. Запускає функціональні тести  
4. Перевіряє стиль коду (напр. PHP_CodeSniffer)  
5. Повідомляє результат: ✓ успішно чи ✗ помилка
```

Якщо CI виявляє помилку, розробник дізнається про неї протягом хвилин (поки контекст зміни ще свіжий у пам'яті), а не через тиждень, коли доведеться заново розбиратися, що саме й чому було змінено. Це і є ключова економічна перевага CI — дешевше виправити помилку одразу, ніж пізніше.

3. Етапи типового CI/CD пайплайну

Пайплайн (pipeline) — послідовність автоматизованих кроків, через які проходить кожна зміна коду. Хоча деталі відрізняються між проєктами, більшість пайплайнів для PHP-застосунку містять подібний набір етапів.

Етап	Що відбувається
Build (збірка)	Встановлення залежностей (composer install), збірка Docker-образу (тема 15)
Test (тестування)	Запуск unit- і функціональних тестів (тема 11)
Lint / Analyze	Перевірка стилю коду та статичний аналіз (пошук потенційних помилок без запуску)
Package	Формування фінального артефакту — готового Docker-образу з номером версії
Deploy (розгортання)	Доставка артефакту на сервер і запуск нової версії застосунку

Якщо будь-який етап завершується невдачею (наприклад, провалився тест), пайплайн зупиняється й не переходить до наступних кроків — це запобігає потраплянню неперевіреного, потенційно зламаного коду на production.

4. GitHub Actions: практичний workflow для PHP

GitHub Actions — одна з найпопулярніших CI/CD-систем, вбудована безпосередньо в GitHub: пайплайн (тут званий workflow) описується декларативно у файлі YAML, що зберігається в самому репозиторії за шляхом `.github/workflows/`.

```
# .github/workflows/ci.yml
name: CI

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Встановити PHP
        uses: shivammathur/setup-php@v2
        with:
          php-version: '8.3'

      - name: Встановити залежності
        run: composer install --no-progress

      - name: Запустити тести
        run: vendor/bin/phpunit
```

Ключ YAML	Призначення
on	Подія, що запускає workflow (push, pull_request тощо)
jobs	Один або кілька незалежних наборів кроків (job), що можуть виконуватись паралельно
runs-on	Тип віртуальної машини, на якій виконується job
steps	Послідовність кроків усередині job — виконуються по черзі
uses	Готовий, повторно використовуваний крок з реєстру GitHub Actions
run	Довільна команда командного рядка

on: push: branches: [main] і pull_request: branches: [main] означають, що workflow запускається як при прямому пуші в гілку main, так і при створенні/оновленні pull request у неї — це дозволяє перевіряти зміни ще до того, як вони потраплять у основну гілку, прямо в інтерфейсі pull request на GitHub.

5. Continuous Delivery vs Continuous Deployment

Після успішного проходження CI (збірка й тести пройшли) постає питання: що робити далі? Тут і розходяться два підходи, які часто плутають через однакову аббревіатуру CD.

Підхід	Як розгортається на production
Continuous Delivery	Автоматично готується до розгортання (напр. Docker-образ зібрано й перевірено), але фактичне розгортання ініціює людина натисканням кнопки
Continuous Deployment	Розгортається на production повністю автоматично одразу після успішного проходження всіх перевірок, без втручання людини

```
# Продовження workflow: розгортання лише для Continuous Deployment
deploy:
  needs: test          # виконується лише якщо job "test" пройшов
  runs-on: ubuntu-latest
  if: github.ref == 'refs/heads/main'
  steps:
    - name: Розгорнути на сервер
      run: |
        docker build -t myapp:${{ github.sha }} .
        docker push myregistry/myapp:${{ github.sha }}
        ssh user@server 'docker pull myregistry/myapp:latest && docker
compose up -d'
```

Вибір між Delivery і Deployment — не технічне, а організаційне рішення: команди, що обслуговують критичні системи (банківські застосунки, медичне обладнання), часто свідомо обирають Continuous Delivery з ручним підтвердженням останнього кроку, тоді як команди, що працюють над менш критичними вебсервісами, дедалі частіше обирають повністю автоматичний Continuous Deployment заради швидкості.

6. Стратегії розгортання: rolling update, blue-green

Просте розгортання (зупинити стару версію, запустити нову) призводить до простою застосунку (downtime), навіть якщо коротко. Для мікросервісів (тема 12), масштабованих горизонтально (тема 14) на кілька екземплярів, існують стратегії, що уникають цього простою.

6.1. Rolling update (послідовне оновлення)

Замість одночасної заміни всіх екземплярів сервісу, нові екземпляри запускаються поступово: спочатку замінюється один старий екземпляр новим, після перевірки його працездатності — наступний, і так по черзі. У будь-який момент часу частина трафіку обробляється старою версією, частина — новою, але сервіс загалом залишається доступним.

6.2. Blue-Green Deployment

Підтримуються дві повністю ідентичні копії production-середовища — «синя» (поточна, що обслуговує весь трафік) і «зелена» (нова версія). Нову версію розгортають і повністю тестують у «зеленому» середовищі, не зачіпаючи «синю», яке продовжує обслуговувати користувачів. Коли «зелена» версія підтверджена як готова, балансувальник навантаження (тема 14) миттєво перемикає весь трафік на неї. Якщо щось піде не так, можна так само миттєво перемкнути назад на «синю» версію.

Стратегія	Перевага	Недолік
Rolling update	Не потребує подвійних ресурсів	Певний час дві версії працюють одночасно
Blue-Green	Миттєве перемикання й миттєвий відкат	Потребує вдвічі більше серверних ресурсів під час розгортання

Обидві стратегії покладаються на ту саму інфраструктурну основу, розглянуту в темі 14: балансувальник навантаження, що розподіляє трафік між кількома екземплярами сервісу, і stateless-архітектуру сервісів, яка дозволяє будь-якому екземпляру (старому чи новому) обробити будь-який запит без різниці для користувача.

7. Повний приклад: від коміту до production

Зберемо все розглянуте в один наскрізний сценарій — шлях однієї зміни коду від розробника до працюючого production-застосунку.

1. Розробник вносить зміну в код (наприклад, виправляє помилку в ProductController, тема 9) і виконує git push у гілку feature-branch.
2. Розробник відкриває pull request у гілку main. GitHub Actions автоматично запускає CI: встановлює залежності, запускає PHPUnit-тести (тема 11).
3. Якщо тести не проходять, pull request позначається як "не пройшов перевірку" — розробник бачить помилку прямо в інтерфейсі GitHub і виправляє код, не чекаючи ручної перевірки колегою.
4. Після успішного проходження CI й схвалення колегою (code review) pull request зливається (merge) у гілку main.
5. Злиття в main запускає наступний workflow: збірка Docker-образу (тема 15) з новим кодом і публікація його в реєстрі образів.
6. Етап розгортання (Continuous Deployment) автоматично оновлює запущені контейнери на сервері через rolling update, поступово замінюючи старі екземпляри новими без простою застосунку.

Цей наскрізний приклад демонструє, як практично кожна тема курсу — тестування (11), контейнеризація (15), балансування навантаження (14) — об'єднується в єдиний автоматизований конвеєр. Саме тому CI/CD розглядають не як окрему ізольовану технологію, а як «клей», що зв'язує всі інші інженерні практики бекенд-розробки в цілісний робочий процес.

Питання для самоперевірки

- Яку конкретну проблему ручного розгортання вирішує автоматизація CI/CD?
- У чому головна мета Continuous Integration — виявити помилки якомога швидше, чи якомога рідше? Обґрунтуйте.
- Перелічіть основні етапи типового CI/CD пайплайну в порядку виконання.
- Що станеться з пайплайном, якщо етап тестування завершиться невдачею? Чому це важливо?
- Чим принципово відрізняється Continuous Delivery від Continuous Deployment?
- Наведіть приклад типу застосунку, для якого команда, ймовірно, обере Continuous Delivery, а не Continuous Deployment.
- Опишіть, як rolling update дозволяє уникнути простою застосунку під час оновлення.
- У чому перевага Blue-Green Deployment над rolling update і якою ціною вона досягається?

Короткий глосарій

Термін	Означення
CI (Continuous Integration)	Автоматична збірка й тестування коду при кожній інтеграції змін
CD (Delivery/Deployment)	Автоматична підготовка до розгортання чи повністю автоматичне розгортання
Пайплайн (pipeline)	Послідовність автоматизованих етапів обробки зміни коду
Workflow (GitHub Actions)	Опис пайплайну у форматі YAML, що зберігається в репозиторії
Job	Незалежний набір кроків усередині workflow
Rolling update	Поступова заміна старих екземплярів сервісу новими без простою
Blue-Green Deployment	Розгортання нової версії в паралельному середовищі з миттєвим перемиканням трафіку
Downtime	Період недоступності застосунку для користувачів