

ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
КАФЕДРА ІНФОРМАТИКИ І КОМП'ЮТЕРНИХ НАУК
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ

КОНСПЕКТ ЛЕКЦІЙ

**Розгортання інформаційних систем (Архітектурні методики та архітектура
програмних додатків у інформаційних системах)**

для здобувачів	Третього курсу
підготовки	першого (бакалаврського) рівня
галузі знань	F (12) Інформаційні технології
спеціальності	F3 (122) Комп'ютерні науки
освітньо-професійної програми	Комп'ютерні науки
факультету	Інформаційних технологій та дизайну

Хмельницький – 2026р.

Конспект лекцій з дисципліни **Розгортання інформаційних систем (Архітектурні методики та архітектура програмних додатків у інформаційних системах)** фахівців на першому бакалаврському рівні вищої освіти спеціальності: F3 (122) Комп'ютерні науки

РОЗРОБНИКИ: Наталя Корніловська к.т.н., доц.

РЕЦЕНЗЕНТ: Світлана Вишемирська к.т.н., доц.

Затверджено на засіданні кафедри І та КН

Протокол №6 від 10.03.2026

В.о. зав. каф. І та КН к.м.н., доц. _____ Світлана Моїсеєнко

Відповідальний за випуск Наталя Корніловська к.т.н., доц.

Узгоджено з навчально-методичним відділом

Реєстраційний номер № _____

ВСТУП

Сучасна розробка програмного забезпечення давно вийшла за межі написання коду. Скласти програму, яка вирішує задачу, це лише половина роботи. Друга, не менш важлива половина, полягає в тому, щоб ця програма запрацювала у реальному світі: була доступна користувачам, стабільно функціонувала під навантаженням, автоматично оновлювалась і безпечно зберігала дані. Саме цим займається дисципліна «Розгортання інформаційних систем».

Конспект лекцій, який ви тримаєте в руках, охоплює весь сучасний стек технологій, що забезпечують шлях програмного продукту від локального комп'ютера розробника до хмарного сервера з реальними користувачами. Матеріал структурований у шість логічно пов'язаних лекцій, кожна з яких розкриває окремий рівень цього шляху.

Лекція 1 формує фундаментальне розуміння самого поняття розгортання: що це таке, чому воно нерозривно пов'язане з хмарними технологіями, як за сорок років галузь еволюціонувала від фізичних серверів у серверних залах до глобальних хмарних платформ. Студент отримує цілісну картину ринку та усвідомлює практичну цінність знань для власної кар'єри.

Лекція 2 заглиблюється у три ключові моделі хмарних сервісів (IaaS, PaaS та SaaS) і вчить головному практичному питанню: що обрати для конкретного завдання. Правильний вибір моделі безпосередньо впливає на вартість, складність і масштабованість системи.

Лекція 3 розкриває принципи побудови сучасних хмарних додатків, зокрема Serverless-архітектуру, де розробник повністю звільнений від управління серверами і зосереджується виключно на логіці застосунку.

Лекція 4 знайомить зі щоденним інструментарієм розробника: системою контролю версій Git, платформою GitHub та хостинговим сервісом Netlify. Студент навчається автоматично публікувати сайти у хмарі через механізм CI/CD, коли кожен git push автоматично оновлює живий сайт.

Лекція 5 присвячена фундаментальним принципам взаємодії між програмними системами: протоколу HTTP, архітектурному стилю REST, формату обміну даними JSON та побудові API на Node.js. Без розуміння цього матеріалу неможливо створити жоден сучасний веб-сервіс.

Лекція 6 завершує курс темою контейнеризації з Docker, технологією, що вирішує одну з найбільш болючих проблем розробки: «у мене на комп'ютері працює, а на сервері ні». Студент опановує Docker, Dockerfile, Docker Compose та принципи розгортання багатокomпонентних систем.

Варто підкреслити, що теми конспекту не є ізольованими розділами, а утворюють єдиний неперервний ланцюг. Git і GitHub (Лекція 4) є передумовою для розуміння CI/CD і Docker (Лекція 6). REST API (Лекція 5) неможливо осмислено розгорнути без розуміння хмарних моделей (Лекції 1-3). Docker-контейнери запускаються на хмарній інфраструктурі, яку описують перші лекції. Читати конспект послідовно є не формальною вимогою, а практичною необхідністю: кожна наступна тема спирається на попередню.

Конспект витриманий у практично орієнтованому стилі: кожна тема не просто пояснюється, а й ілюструється реальними сценаріями з індустрії, конкретними наслідками помилок і готовими прикладами коду та конфігурацій. Окремий акцент зроблено на типових помилках, і не для залякування, а для формування інженерної відповідальності, адже помилки при розгортанні у реальних системах можуть призводити до втрати даних, фінансових збитків і компрометації безпеки. Наведені у конспекті реальні інциденти (студент, якому виставили рахунок на \$47 000 за ніч через випадково опублікований API-ключ; команди, що витрачали сотні годин на рік через відсутність Docker) є задокументованими випадками, що стаються з реальними розробниками щодня. Знання, викладені у цьому конспекті, безпосередньо захищають від таких ситуацій.

Важливо також зазначити контекст часу, в якому написаний цей матеріал. Станом на 2026 рік штучний інтелект стрімко змінює ландшафт розробки: інструменти на кшталт GitHub Copilot вже автоматично генерують значну частину коду. Проте розгортання, інфраструктура та DevOps-практики

залишаються тією областю, де розуміння принципів, а не лише уміння скопіювати команду, є принциповою конкурентною перевагою. Автоматизовані інструменти можуть написати Dockerfile, але не можуть вирішити, яку архітектуру обрати, чому упав деплой або як правильно ізолювати мережі контейнерів. Ці рішення залишаються за людиною.

Матеріал розрахований на здобувачів третього курсу першого (бакалаврського) рівня підготовки за спеціальністю 122 «Комп'ютерні науки» Херсонського національного технічного університету. Для успішного засвоєння достатньо базових знань з програмування та загального розуміння того, як працює Інтернет. Усі ключові терміни англійською мовою подаються з перекладом та поясненням, оскільки технічна документація, Stack Overflow, курси та більшість ресурсів галузі існують переважно англійською, і вміння впевнено читати технічні тексти є окремою професійною навичкою, яку цей конспект також розвиває.

Після опанування курсу студент буде здатний розгорнути повноцінний веб-застосунок у хмарі, налаштувати автоматичний CI/CD-пайплайн, спроектувати REST API, упакувати систему у Docker-контейнери та впевнено спілкуватися з DevOps-інженерами і технічними рекрутерами на одній мові, мові сучасного промислового розгортання. А найголовніше, отримати не набір розрізнених команд для заучування, а цілісне розуміння того, як сучасні інформаційні системи живуть, дихають і працюють у реальному світі.

Зміст

ВСТУП	3
ЛЕКЦІЯ 1. Що таке розгортання інформаційних систем.....	7
хмарні обчислення: еволюція, моделі та провайдери	7
ЛЕКЦІЯ 2. Моделі хмарних сервісів: iaas, paas та saas - що обрати і коли	26
ЛЕКЦІЯ 3. АРхітектура та принципи хмарних додатків: Serverless-архітектура та принципи хмарних додатків	41
ЛЕКЦІЯ 4. GitHub та Netlify - сучасні інструменти розробника	54
ЛЕКЦІЯ 5. API · REST · HTTP · JSON · Node.js · CRUD.....	77
Лекція 6. КОНТЕЙНЕРИЗАЦІЯ ДОДАТКІВ З DOCKER. СТВОРЕННЯ ТА РОЗГОРТАННЯ БАГАТОКОМПОНЕНТНИХ СИСТЕМ	102
РЕКОМЕНДОВАНА ЛІТЕРАТУРА	119

ЛЕКЦІЯ 1. Що таке розгортання інформаційних систем. хмарні обчислення: еволюція, моделі та провайдери

Мета лекції

Дати студентам цілісне уявлення про дисципліну «Розгортання інформаційних систем»: пояснити, що таке розгортання (deployment) як окремий етап життєвого циклу програмного забезпечення і чому сьогодні воно нерозривно пов'язане з хмарними обчисленнями. Сформулювати розуміння концепції хмарних обчислень, їхнє офіційне визначення та п'ять ключових характеристик за стандартом NIST (National Institute of Standards and Technology - Національний інститут стандартів і технологій США). Ознайомити з еволюцією підходів до розгортання ІС (інформаційних систем) - від фізичних серверів до сучасних хмарних платформ. Показати місце хмарних знань у кар'єрі ІТ-фахівця спеціальності 122 «Комп'ютерні науки».

План лекції

1. Що таке розгортання інформаційних систем і чому воно стало хмарним.
2. Навіщо це знати: місце хмарних технологій у кар'єрі ІТ-фахівця.
3. Еволюція розгортання ІС: від фізичних серверів до хмари.
4. Що таке хмарні обчислення: визначення NIST, п'ять ключових характеристик.
5. Основні хмарні провайдери та моделі розгортання хмари.

ЩО ТАКЕ РОЗГОРТАННЯ ІНФОРМАЦІЙНИХ СИСТЕМ І ЧОМУ ВОНО СТАЛО ХМАРНИМ

Визначення розгортання

Розгортання (deployment - від англ. «розгорнути, розмістити») інформаційних систем - це сукупність процесів, дій і технологій, що забезпечують перенесення готового програмного забезпечення з середовища розробки у виробниче середовище, де воно стає доступним для кінцевих користувачів.

Простіше кажучи: розробник написав і протестував програму на своєму комп'ютері. Але поки вона там - нею може користуватися лише сам розробник. Розгортання - це процес, завдяки якому ця програма «з'являється» в Інтернеті або корпоративній мережі та починає приймати запити від реальних користувачів. Без розгортання будь-який, навіть найгеніальніший код залишається мертвим набором файлів на локальному диску.

Важливо розуміти: розгортання - це не просто «залити файли на сервер». Це комплексний інженерний процес, що охоплює підготовку інфраструктури,

налаштування середовища виконання, конфігурацію мережі та безпеки, тестування у виробничому середовищі, запуск і безперервний моніторинг після запуску. Помилки при розгортанні - одна з найбільш поширених причин простоїв і втрат бізнесу.

Наочний приклад: команда розробила онлайн-магазин. Вони тижнями писали код, тестували, виправляли помилки. Нарешті, продукт готовий. Що відбувається далі?

- ✓ Необхідно обрати та підготувати сервер - фізичний або хмарний.
- ✓ Налаштувати операційну систему, веб-сервер (Nginx або Apache - популярні веб-сервери для роздавання сторінок), базу даних.
- ✓ Завантажити код застосунку на сервер та встановити всі залежності (бібліотеки, пакети).
- ✓ Налаштувати доменне ім'я (наприклад, myshop.com), SSL-сертифікат (Secure Sockets Layer / Transport Layer Security - протокол шифрування з'єднання, що забезпечує HTTPS).
- ✓ Налаштувати систему моніторингу - щоб одразу дізнаватись, якщо щось перестане працювати.
- ✓ Відкрити доступ для користувачів і відстежувати перші реальні запити.

Усі ці кроки і є розгортанням. І якщо раніше кожен з них виконувався вручну системним адміністратором протягом кількох днів або тижнів, то сучасні хмарні платформи дозволяють автоматизувати весь цей процес до кількох хвилин або навіть секунд.

Місце розгортання в життєвому циклі програмного забезпечення

Розгортання є одним із ключових етапів SDLC (Software Development Life Cycle - Життєвий цикл розробки програмного забезпечення). Розглянемо класичну послідовність:

Етап SDLC	Що відбувається	Хто відповідає
1. Планування (Planning)	Визначення вимог, бюджету, строків	Менеджер проекту, замовник
2. Аналіз вимог (Requirements Analysis)	Деталізація функціональних та нефункціональних вимог	Аналітик, замовник
3. Проектування (Design)	Архітектура системи, вибір технологій	Архітектор, старший розробник
4. Розробка (Development)	Написання коду, unit-тестування (тестування окремих компонентів)	Розробники
5. Тестування (Testing)	QA (Quality Assurance - забезпечення якості): перевірка на помилки	QA-інженери

Етап SDLC	Що відбувається	Хто відповідає
6. РОЗГОРТАННЯ (Deployment)	Перенесення у виробниче середовище, налаштування інфраструктури	DevOps-інженери, розробники
7. Підтримка (Maintenance)	Моніторинг, оновлення, виправлення помилок	DevOps, розробники

Зверніть увагу: раніше розробка і розгортання вважалися двома різними, чітко розділеними областями відповідальності. Розробники писали код і «передавали його через стіну» системним адміністраторам, які займалися розгортанням. Цей підхід породжував конфлікти, затримки та помилки. Саме для подолання цього розриву і виникла концепція DevOps (Development + Operations - Розробка + Операції) - культура і набір практик, що об'єднують команди розробки та адміністрування у єдиний процес неперервної інтеграції та доставки.

Чому розгортання стало хмарним

Це ключове питання: якщо розгортання існувало ще до появи хмарних технологій, то чому сьогодні ці поняття практично нерозривні?

Відповідь криється у самій природі хмари. Хмарні платформи - AWS (Amazon Web Services - хмарні вебсервіси Amazon), Azure (Microsoft Azure - хмарна платформа Microsoft), GCP (Google Cloud Platform - хмарна платформа Google) - надають готову, масштабовану, керовану інфраструктуру через Інтернет. Розробник або DevOps-інженер більше не займається купівлею серверів, облаштуванням серверної кімнати або ручним встановленням операційних систем. Замість цього - кілька команд у терміналі або кліків у веб-консолі, і нова інфраструктура готова.

Сучасний процес розгортання виглядає так: розробник вносить зміни до коду і виконує git push (команда відправки змін у систему контролю версій Git). Це автоматично запускає CI/CD-пайплайн (CI - Continuous Integration - неперервна інтеграція; CD - Continuous Delivery / Deployment - неперервна доставка / розгортання). Система автоматично збирає проєкт, запускає тести, і якщо все успішно - розгортає нову версію у хмарі. Весь цикл займає від 2 до 10 хвилин. Без хмари - той самий процес міг займати дні.

Три фундаментальні причини, чому розгортання перейшло у хмару:

Перша причина - швидкість. Традиційне розгортання на фізичних серверах могло займати тижні: закупівля обладнання, доставка, монтаж, налаштування. Хмара скорочує цей час до хвилин. Для сучасного бізнесу, де конкуренти випускають оновлення щодня, це критично важливо.

Друга причина - масштабованість. Фізичний сервер має фіксовану потужність. Якщо завтра на сайт прийде у 100 разів більше користувачів - він «впаде». Хмара автоматично масштабується: додає ресурси при зростанні навантаження і вивільняє їх після спаду. Це неможливо реалізувати на власних серверах без колосальних фінансових витрат.

Третя причина - економічна модель. Власні сервери - це капітальні витрати (CapEx - Capital Expenditure): велика сума грошей одразу, незалежно від того, чи використовується обладнання. Хмара - це операційні витрати (OpEx - Operational Expenditure): оплата за фактичне використання. Стартап може розпочати з безкоштовного тарифу і масштабуватись у міру зростання, не вкладаючи мільйони в інфраструктуру на старті.

Отже, розгортання інформаційних систем стало хмарним не тому, що «так модно», а тому, що хмара вирішує реальні, фундаментальні проблеми традиційного підходу. Ця дисципліна навчить вас розгортати сучасні системи саме так, як це робиться у реальній індустрії.

НАВІЩО ЦЕ ЗНАТИ: МІСЦЕ ХМАРНИХ ТЕХНОЛОГІЙ У КАР'ЄРІ ІТ-ФАХІВЦЯ

Ринок праці та хмарні технології

Хмарні обчислення - це вже не «технологія майбутнього». За даними аналітичної компанії Gartner (провідна дослідницька та консалтингова компанія у сфері ІТ), у 2024 році понад 94 % підприємств у світі використовують принаймні один хмарний сервіс. Обсяг ринку хмарних обчислень перевищив 600 млрд доларів США на рік і продовжує зростати на 20–25 % щороку - значно швидше за решту ІТ-галузі.

Це означає, що абсолютна більшість роботодавців, до яких прийдуть випускники спеціальності 122 «Комп'ютерні науки», вже використовують або активно впроваджують хмарні рішення. Приблизно 80 % вакансій для розробників програмного забезпечення в Україні та Польщі передбачають базові знання хмарних технологій - і ця частка зростає щороку.

Показник ринку	Значення (2024)	Що це означає для вас
Частка підприємств, що використовують хмару	94 %	Майже кожна компанія, куди ви підете на роботу, вже використовує хмарні сервіси
Обсяг світового ринку хмарних обчислень	понад 600 млрд USD/рік	Один з найбільших і найбільш зростаючих секторів ІТ-індустрії
Щорічне зростання ринку	20–25 % на рік	Попит на фахівців зростає швидше за інші ІТ-спеціальності

Показник ринку	Значення (2024)	Що це означає для вас
Вакансії розробників, що вимагають знання хмари	~80 %	Без базових знань хмари ви автоматично відсіваєтесь від більшості вакансій
Середня зарплата AWS Solutions Architect (архітектор хмарних рішень AWS) в Україні	від 3 000 USD/місяць	Хмарна сертифікація суттєво збільшує ринкову вартість спеціаліста

Хмарні знання та конкурентна перевага

Важливо розуміти: вивчення розгортання - це не просто «ще один предмет у програмі». Це формування практичних навичок, які роботодавці перевіряють безпосередньо на технічних інтерв'ю. Запитання на кшталт «Як би ви розгорнули цей застосунок у хмарі?», «Що таке CI/CD?», «Яка різниця між IaaS та PaaS?» - стандартна частина інтерв'ю junior-розробника у 2024 році.

Крім того, хмарні сертифікації - AWS Certified Cloud Practitioner (сертифікований фахівець з хмарних практик AWS), Microsoft AZ-900 (Azure Fundamentals - основи Azure), Google Cloud Associate Cloud Engineer - суттєво підвищують конкурентоспроможність випускника. Більшість з них доступні студентам безкоштовно або зі значними знижками.

Після вивчення цього курсу студент буде здатний:

- ✓ розуміти, що таке розгортання ІС і як воно вписується у повний цикл розробки програмного забезпечення;
- ✓ обирати правильну модель розгортання (IaaS / PaaS / SaaS) для конкретного завдання;
- ✓ розгорнути веб-застосунки у хмарі без фізичних серверів - за лічені хвилини;
- ✓ розуміти, за що відповідає хмарний провайдер, а що залишається відповідальністю розробника;
- ✓ використовувати безкоштовні хмарні ресурси для власних навчальних проєктів і портфоліо;
- ✓ спілкуватися однією мовою з DevOps-інженерами (інженерами з розробки та операцій), архітекторами систем і тімлідами (технічними керівниками команд).

ЕВОЛЮЦІЯ РОЗГОРТАННЯ ІНФОРМАЦІЙНИХ СИСТЕМ: ВІД ФІЗИЧНИХ СЕРВЕРІВ ДО ХМАРИ

Щоб повною мірою зрозуміти цінність хмарних технологій і усвідомити, чому вони стали стандартом індустрії, необхідно розглянути, яким шляхом прийшла галузь до сучасного

стану. Протягом останніх сорока років розгортання інформаційних систем пройшло три принципово різні епохи, кожна з яких вирішувала конкретні проблеми попередньої.

Епоха фізичних серверів - Bare-metal (до 2000-х років)

Уявімо ситуацію 1998 року: компанія хоче запустити корпоративний веб-сайт. Що для цього потрібно? По-перше - придбати фізичний сервер, тобто окремий комп'ютер з дуже потужними характеристиками, призначений для обробки запитів багатьох користувачів одночасно. Вартість такого обладнання - від 5 000 до 50 000 доларів США залежно від продуктивності. По-друге - очікувати доставки та монтажу: від замовлення до готового до роботи сервера проходило 4–12 тижнів. По-третє - облаштувати окремий серверний зал з кондиціонуванням, безперебійним живленням (UPS - Uninterruptible Power Supply - джерело безперебійного живлення: пристрій, що забезпечує роботу обладнання під час відключення електроенергії) та фізичною охороною. По-четверте - найняти системного адміністратора на постійну ставку.

Термін «bare-metal» буквально означає «голе залізо» - тобто один застосунок безпосередньо на одному фізичному сервері без жодного програмного «прошарку» між ними.

Найгостріша проблема епохи - катастрофічна неефективність використання ресурсів. Дослідження показують: середній корпоративний сервер того часу використовував лише 10–20 % своєї обчислювальної потужності. Решта 80–90 % - просто простояла, продовжуючи споживати електроенергію та потребувати обслуговування.

Проблема масштабованості була критичною. Розглянемо типовий сценарій: стартап очікує 10 000 одночасних користувачів і купує відповідний сервер. Якщо реальний старт привернув лише 500 користувачів - кошти витрачені, ресурси простоюють. Якщо ж сервіс несподівано «злетів» і зібрав 100 000 активних користувачів - сервер «падає» під навантаженням. Докупити потужностей швидко неможливо: строк від замовлення до готового сервера - кілька тижнів. Бізнес втрачає клієнтів і репутацію.

Інша типова ситуація: роздрібний магазин розгортає систему онлайн-продажів перед різдвяними розпродажами. Вони купують сервер, розрахований на пікове навантаження - 50 000 транзакцій на день. Але пік триває лише 2 тижні на рік. Решта 50 тижнів - сервер стоїть і споживає електроенергію, обробляючи 3 000 транзакцій на день.

Епоха віртуалізації (2000–2010 роки)

Вирішення проблеми неефективності прийшло з технологією віртуалізації. Суть ідеї елегантна: один потужний фізичний сервер можна програмно «поділити» на кілька незалежних VM (Virtual Machine - Віртуальна Машина) - програмних імітацій окремих комп'ютерів. Кожна VM «думає», що вона окремий фізичний комп'ютер, має власну операційну систему і ресурси, але насправді ділить фізичне обладнання з іншими VM.

Ключовим компонентом, що уможливив це, став гіпервізор (Hypervisor) - спеціальне програмне забезпечення, що управляє VM і розподіляє між ними ресурси фізичного сервера. Найвідоміші гіпервізори: VMware vSphere (корпоративне рішення від VMware), Microsoft Hyper-V (гіпервізор від Microsoft, вбудований у Windows Server), KVM (Kernel-based Virtual Machine - Віртуальна Машина на основі ядра Linux: рішення з відкритим кодом, вбудоване безпосередньо у ядро Linux).

Переваги були суттєвими: ефективність використання обладнання зросла до 60–80 % - на одному фізичному сервері тепер могли одночасно працювати 10–20 незалежних VM. Новий «сервер» можна запустити за хвилини, а не чекати фізичної доставки. Якщо одна VM «впала» через помилку в застосунку - інші продовжують працювати незалежно.

Практичний приклад: університетська обчислювальна лабораторія купує один потужний сервер і запускає на ньому 15 VM - кожна для окремого студентського проекту. Замість купівлі 15 фізичних серверів - один з набагато вищою ефективністю використання ресурсів.

Проте фундаментальна проблема нікуди не зникла: фізичне обладнання все ще купується, обслуговується і рано чи пізно застаріває. Масштабування за межі можливостей наявного залізодосі залишалось болючою проблемою.

Епоха хмарних обчислень (2006 рік - сьогодення)

У 2006 році компанія Amazon запустила сервіс AWS EC2 (Amazon Web Services Elastic Compute Cloud - Еластична хмарна обчислювальна служба Amazon). Ідея була революційною: Amazon вже мала колосальну власну IT-інфраструктуру для обслуговування інтернет-магазину і вирішила «здавати її в оренду» іншим компаніям через Інтернет. Замість купівлі серверів - щогодинна оренда. Замість серверного залу - датацентр (ЦОД - Центр Обробки Даних, або Data Center) провайдера.

Революційний принцип хмари - pay-as-you-go (платіть у міру використання). Аналогія: ніхто не платить за електроенергію «наперед за місяць незалежно від споживання» - лише за фактично спожиті кіловат-години. Так само в хмарі: запустили сервер на дві години - заплатили за дві години. Масштабували до ста серверів під час пікового навантаження - заплатили лише за ці години, після чого система автоматично повернулась до мінімуму.

Дуже точна аналогія для розуміння хмари - електроенергія. На початку XX століття кожен завод мав власний генератор. Заводи наймали інженерів для його обслуговування, витрачали кошти на паливо та ремонт. Коли з'явилися електростанції і централізовані мережі - заводи просто підключились до розетки. Ніхто більше не думає, яка електростанція живить його виробничу лінію. Просто сплачують за кіловати. Хмара зробила те саме з IT-

інфраструктурою: замість власних серверів - підключаєтесь до інфраструктури AWS / Azure / GCP і платите за те, що використовуєте.

Переломний момент для індустрії стався у 2008 році, коли Netflix - на той час сервіс доставки DVD-дисків поштою - прийняв рішення про повний перехід на AWS. Це рішення стало легендарним в індустрії: Netflix пояснив, що AWS дозволяє зосередитись на основному продукті (відеосервісі), а не на управлінні інфраструктурою. Сьогодні Netflix обслуговує понад 260 мільйонів підписників у 190 країнах, використовуючи десятки тисяч серверних інстансів в AWS - і міг би це зробити без єдиного власного сервера.

Порівняльна таблиця трьох епох розгортання ІС:

Характеристика	Фізичні сервери (до 2000-х)	Віртуалізація (2000–2010)	Хмарні обчислення (2006–сьогодні)
Ключова технологія	Bare-metal (голе залізо): один застосунок - один фізичний сервер	Hypervisor (гіпервізор): VMware, Hyper-V, KVM - багато VM на одному залізі	Розподілена віртуалізація у ЦОД провайдера по всьому світу
Ефективність ресурсів	10–20 % (більшість потужності простоє)	60–80 % (значне покращення)	80–95 % (пул ресурсів розподіляється динамічно між тисячами клієнтів)
Час отримання нового сервера	4–12 тижнів (замовлення + доставка + монтаж)	Хвилини (запуск нової VM)	Секунди - хвилини (через API, CLI або веб-консоль)
Масштабування	Неможливе швидко: треба купити нове залізо	Обмежене ресурсами наявного фізичного сервера	Автоматичне: сотні серверів запускаються за лічені хвилини при піковому навантаженні
Модель витрат	CapEx (капітальні витрати): велика сума одразу, незалежно від використання	CapEx + менші витрати на обслуговування	OpEx (операційні витрати): pay-as-you-go, лише за фактичне споживання
Відповідальність за обладнання	Повністю ваша: купівля, монтаж, ремонт, оновлення	Ваша, але ефективніше: одне залізо - багато застосунків	Провайдера: ви не торкаєтесь фізичного обладнання взагалі

Контейнеризація - наступний крок після хмари

Паралельно з розвитком хмарних платформ у 2010-х роках виникла ще одна революційна технологія - контейнеризація. Контейнер (container) - це ізольоване середовище

виконання, що містить застосунок і всі його залежності (бібліотеки, налаштування), але на відміну від VM не включає повноцінну операційну систему. Контейнери набагато легші і швидші за VM.

Найпопулярніший інструмент контейнеризації - Docker (відкрита платформа для розробки, доставки та запуску застосунків у контейнерах, запущена у 2013 році). Якщо VM - це окрема квартира зі своїм ремонтом і меблями, то контейнер - це валіза зі своїми речами, яку можна відкрити у будь-якій квартирі. Застосунок у контейнері однаково працює на ноутбучі розробника, у тестовому середовищі і на виробничому хмарному сервері.

Для управління великою кількістю контейнерів використовується Kubernetes (K8s - скорочено, де 8 - кількість букв між «К» і «s»: відкрита система оркестрації контейнерів, розроблена Google і передана у 2014 році відкритому співтовариству). Kubernetes автоматично запускає, перезапускає, масштабує і балансує навантаження між контейнерами.

Чому це важливо для дисципліни: сьогодні більшість промислових хмарних розгортань використовують контейнери. Розуміння Docker і Kubernetes - одна з найбільш затребуваних навичок DevOps-інженера.

4.

ЩО ТАКЕ ХМАРНІ ОБЧИСЛЕННЯ: ВИЗНАЧЕННЯ NIST І П'ЯТЬ КЛЮЧОВИХ ХАРАКТЕРИСТИК

Офіційне визначення за NIST SP 800-145

Офіційне визначення хмарних обчислень сформулював NIST (National Institute of Standards and Technology - Національний інститут стандартів і технологій США) у документі NIST SP 800-145, опублікованому у 2011 році. Незважаючи на вік документа, це визначення досі є міжнародним еталоном і цитується у більшості академічних і промислових джерел:

«Cloud computing (хмарні обчислення) - це модель забезпечення зручного, повсюдного мережевого доступу до спільного пулу конфігурованих обчислювальних ресурсів (мереж, серверів, сховищ, додатків і сервісів), які можуть бути швидко надані та звільнені з мінімальними управлінськими зусиллями або взаємодією з постачальником послуг.» - NIST SP 800-145, Mell P. and Grance T., 2011.

Розберемо ключові слова визначення:

- ✓ Зручний (convenient) - не потрібно спеціальних технічних знань для отримання ресурсу; веб-інтерфейс провайдера доступний будь-якому фахівцю.
- ✓ Повсюдний (ubiquitous) - доступний з будь-якої точки планети через Інтернет.
- ✓ Спільний пул ресурсів (shared pool) - тисячі серверів провайдера об'єднані у єдиний пул і динамічно розподіляються між клієнтами.

✓ Конфігурованих (configurable) - ресурси можна налаштувати під конкретні потреби: обрати кількість ядер CPU (Central Processing Unit - Центральний процесор), об'єм RAM (Random Access Memory - Оперативна пам'ять), тип і об'єм сховища.

✓ Швидко надані та звільнені (rapidly provisioned and released) - хвилини або секунди, а не тижні.

✓ Мінімальні управлінські зусилля (minimal management effort) - клієнт не повинен вручну налаштовувати фізичне обладнання.

Простіше кажучи: ви отримуєте ІТ-ресурси через Інтернет, платите тільки за те, що використовуєте, і не займаєтесь фізичним обладнанням.

П'ять обов'язкових характеристик хмарних обчислень

NIST виокремлює п'ять обов'язкових характеристик (essential characteristics), без яких сервіс не може вважатися «справжньою хмарою». Це важливо: деякі постачальники використовують слово «хмара» у маркетингових цілях, пропонуючи по суті традиційний хостинг. Знання цих п'яти ознак дозволяє розрізнити справжню хмару від «хмарного прання».

Характеристика 1. On-demand self-service - Самообслуговування на вимогу

Користувач може самостійно запустити або зупинити будь-який хмарний ресурс у будь-який момент без дзвінка до служби підтримки, без погодження з адміністратором і без очікування. Через веб-консоль, CLI (Command Line Interface - інтерфейс командного рядка: текстовий інтерфейс для взаємодії з комп'ютером через введення команд) або API (Application Programming Interface - Програмний інтерфейс застосунків: набір правил і протоколів для взаємодії програм між собою) за 2–5 хвилин доступний новий сервер з обраними характеристиками.

Для порівняння: у «докхмарну» еру отримання нового фізичного сервера займало 4–12 тижнів: замовлення, доставка, монтаж у стійку (rack - стелаж для монтажу серверного обладнання), підключення до мережі, встановлення ОС (Операційна Система, або OS - Operating System: системне програмне забезпечення, що управляє апаратним забезпеченням і надає базові послуги іншим програмам). Тепер - кілька кліків у консолі, і IP-адреса (Internet Protocol address - унікальна адреса пристрою в мережі Інтернет) нового сервера готова.

Практичний приклад: стартап отримав інвестиції і хоче терміново масштабувати серверну потужність перед запуском маркетингової кампанії. О 23:00 у п'ятницю. Без хмари - неможливо. З хмарою AWS - DevOps-інженер заходить у консоль, обирає потрібний тип інстансу (instance - окремий хмарний сервер), натискає «Launch», і через 3 хвилини новий сервер вже приймає трафік.

Характеристика 2. Broad network access - Широкий мережевий доступ

Хмарні ресурси доступні з будь-якого пристрою (ноутбук, смартфон, планшет, IoT-пристрій - Internet of Things - Інтернет речей: пристрої з підключенням до Інтернету, не призначені спеціально для обчислень, наприклад розумний термостат або виробничий датчик) і з будь-якої точки планети через стандартні мережеві протоколи.

Не потрібен VPN (Virtual Private Network - Віртуальна приватна мережа: технологія шифрованого тунелю між двома мережами) до корпоративної мережі, не потрібно фізично знаходитись в офісі. Виробничий сервер (production server - сервер, на якому працює реальний застосунок для реальних користувачів) доступний однаково добре з Одеси, Варшави або Токіо.

Саме ця характеристика стала ключовою під час пандемії COVID-19 у 2020 році: компанії, що вже мали хмарну інфраструктуру, перевели співробітників на дистанційний режим за лічені дні. Ті, хто покладались виключно на власні корпоративні сервери, зіткнулись з серйозними проблемами доступу.

Характеристика 3. Resource pooling - Об'єднання ресурсів у спільний пул

Провайдер об'єднує фізичні ресурси тисяч серверів у єдиний пул і динамічно розподіляє їх між клієнтами залежно від поточного попиту. Кінцевий користувач не знає і не може знати, на якому конкретному фізичному сервері у якому датацентрі виконується його код - і в цьому немає потреби.

Ця концепція називається multi-tenancy (мультиоренда - модель, за якої один програмний або апаратний ресурс обслуговує кількох клієнтів, повністю ізолюючи їхні дані і процеси одне від одного). Кожен клієнт «думає», що використовує виділені ресурси, хоча насправді вони розподілені між багатьма.

Економічний наслідок: провайдер закуповує обладнання оптом у величезних кількостях - серверні зали AWS і Google містять мільйони фізичних серверів. Завдяки масштабу закупівель ціна за одиницю обчислень для кінцевого клієнта виявляється нижчою, ніж якщо б він купував власне обладнання.

Характеристика 4. Rapid elasticity - Швидка еластичність

Ресурси можна масштабувати вгору і вниз автоматично залежно від навантаження. Розрізняють два типи масштабування:

- ✓ Scale up / Vertical scaling (вертикальне масштабування) - збільшення потужності окремого сервера: більше CPU-ядер, більше RAM. Аналогія: замінити малолітражний автомобіль на автомобіль з більшим двигуном.
- ✓ Scale out / Horizontal scaling (горизонтальне масштабування) - додавання нових серверів до кластера (cluster - група серверів, що працюють спільно). Аналогія: замість одного великого автомобіля - парк з десяти менших. Горизонтальне масштабування більш притаманне хмарним архітектурам.

Класичний приклад: інтернет-магазин Rozetka.ua під час розпродажу «Чорна п'ятниця» (Black Friday) або перед Новим роком отримує у 10–50 разів більше трафіку, ніж у звичайний день. Хмара автоматично запускає додаткові сервери при зростанні навантаження (auto-scaling - автоматичне масштабування) і вимикає їх, коли хвиля спадає. Оплата - лише за реально використані ресурси у часи пікового навантаження. Без хмари компанія мала б або купити обладнання «на пік» і воно простоювало б 350 днів на рік, або ризикувати «падінням» у найважливіший комерційний день.

Ще один приклад: стриміновий сервіс анонсує прем'єру популярного серіалу. Мільйони глядачів підключаються одночасно. Хмара за лічені хвилини додає сотні серверів для обробки навантаження - і після першого «ажіотажного» тижня автоматично вивільняє їх.

Характеристика 5. Measured service - Вимірювана послуга (Pay-as-you-go)

Хмарні ресурси вимірюються і тарифікуються з точністю до секунди, а деякі сервіси - до мілісекунди. Ви платите за конкретні виміряні показники: vCore-години (одиниці обчислювального часу: 1 vCore-година - це 1 віртуальне ядро CPU, що працює 1 годину), гігабайти переданих даних, кількість запитів до бази даних, секунди роботи серверних функцій.

Модель AWS Lambda (безсерверні функції - Serverless Functions: модель виконання коду, за якої сервер не потрібен постійно, а функція запускається лише при надходженні запиту і вимикається одразу після виконання) тарифікує за кількість запитів і мілісекунди часу виконання. Якщо функція виконується 200 мс і викликається 1 мільйон разів на місяць - ви платите саме за ці $200 \text{ мс} \times 1\,000\,000$, а не за «сервер, який стоїть увімкненим».

Аналогія з комунальними послугами: ніхто не сплачує за воду «наперед за місяць незалежно від споживання» - лише за кубометри, що реально витекли з крана. Провайдери надають детальні звіти про споживання і інструменти для виявлення «непотрібних» витрат - наприклад, AWS Cost Explorer або Google Cloud Billing.

Зведена таблиця п'яти ключових характеристик хмари:

Характеристика NIST	Англійський термін	Практичний зміст	Приклад
Самообслуговування на вимогу	On-demand self-service	Сервер доступний за кілька хвилин без участі адміністратора провайдера	О 3:00 ночі DevOps-інженер запускає новий сервер через веб-консоль AWS
Широкий мережевий доступ	Broad network access	Хмарні ресурси доступні з будь-якого пристрою і місця через Інтернет	Студент керує хмарним сервером з мобільного телефону

Характеристика NIST	Англійський термін	Практичний зміст	Приклад
Пул ресурсів	Resource pooling	Фізичні ресурси тисяч серверів об'єднані і динамічно розподіляються між клієнтами	Один і той самий фізичний сервер AWS обслуговує клієнтів з різних країн з повною ізоляцією даних
Швидка еластичність	Rapid elasticity	Ресурси автоматично масштабуються вгору і вниз залежно від навантаження	Сайт автоматично збільшує кількість серверів під час «Чорної п'ятниці» і зменшує після
Вимірювана послуга	Measured service / Pay-as-you-go	Оплата за фактичне використання: секунди, гігабайти, кількість запитів	Функція AWS Lambda тарифікується за мілісекунди реального виконання, а не за «включений сервер»

ОСНОВНІ ХМАРНІ ПРОВАЙДЕРИ ТА МОДЕЛІ РОЗГОРТАННЯ ХМАРИ

Три провідні хмарні провайдери

Сьогодні ринок хмарних обчислень очолюють три великих гравці: AWS (Amazon Web Services - хмарні вебсервіси Amazon), Microsoft Azure (хмарна платформа Microsoft) та GCP (Google Cloud Platform - хмарна платформа Google). Разом вони займають близько 65 % світового ринку хмарних обчислень. Розглянемо кожного детально.

AWS - Amazon Web Services (хмарні вебсервіси Amazon)

AWS - безперечний лідер ринку з часткою близько 32 %. Запущений у 2006 році, сьогодні пропонує понад 200 різних послуг: від базових віртуальних машин до квантових обчислень. Ключові сервіси: EC2 (Elastic Compute Cloud - еластична хмарна обчислювальна служба: віртуальні машини), S3 (Simple Storage Service - простий сервіс зберігання: об'єктне сховище для файлів будь-якого типу), Lambda (серверні функції: виконання коду без потреби управляти сервером), RDS (Relational Database Service - сервіс реляційних баз даних: керована СУБД - Система Управління Базами Даних).

Серед клієнтів AWS - Netflix, Airbnb, NASA (National Aeronautics and Space Administration - Національне управління з аеронавтики і дослідження космічного простору), Spotify, Twitch. AWS має найбільшу спільноту і найбільшу кількість навчальних матеріалів. Для студентів - безкоштовний рівень AWS Free Tier: 12 місяців безкоштовно (включаючи t2.micro VM - мала віртуальна машина і 5 ГБ S3-сховища). Посилання: aws.amazon.com/free

Microsoft Azure (хмарна платформа Microsoft)

Azure займає близько 23 % ринку і є природним вибором для компаній, що використовують Microsoft-стек: .NET (платформа розробки від Microsoft), SQL Server, Windows Server, Active Directory (AD - служба каталогів Microsoft для управління ідентифікацією і доступом). Запущений у 2010 році. Ключові сервіси: Azure Virtual Machines (VM - віртуальні машини), Azure App Service (хостинг веб-застосунків: PaaS-рішення для публікації застосунків без управління сервером), Azure SQL Database (керована хмарна СУБД), Azure Functions (безсерверні функції), Azure Active Directory (хмарна служба управління ідентифікацією).

Серед клієнтів - BMW, Adobe, Walmart, eBay, Samsung. Для студентів: програма Azure for Students - 100 USD кредит без кредитної картки. Посилання: azure.microsoft.com/free/students

GCP - Google Cloud Platform (хмарна платформа Google)

GCP займає близько 10 % ринку, але є лідером у сфері великих даних і ML (Machine Learning - Машинне навчання: розділ штучного інтелекту, що дозволяє системам навчатися на даних без явного програмування). Запущений у 2008 році. Ключові сервіси: Compute Engine (VM - обчислювальні ресурси), BigQuery (сервіс аналізу великих даних: обробка петабайтів даних за секунди), Cloud Run (запуск контейнерів без управління сервером), Vertex AI (платформа для розробки і розгортання ML-моделей).

Серед клієнтів - Spotify, Twitter, Snapchat, Etsy. Для студентів: 300 USD кредит на 90 днів. Посилання: cloud.google.com/free

Провайдер	Частка ринку	Сильна сторона	Ключові сервіси	Безкоштовно для студентів
AWS (Amazon Web Services)	~32 % (лідер)	Найширший вибір сервісів, найбільша спільнота, найбільше навчальних матеріалів	EC2, S3, Lambda, RDS, CloudFront	Free Tier: 12 міс. aws.amazon.com/free
Microsoft Azure	~23 % (2-ге місце)	Інтеграція з Microsoft-продуктами, корпоративний сектор	Virtual Machines, App Service, Azure SQL, Functions, Active Directory	100 USD кредит. azure.microsoft.com/free/students

Провайдер	Частка ринку	Сильна сторона	Ключові сервіси	Безкоштовно для студентів
GCP (Google Cloud Platform)	~10 % (3-тє місце)	Великі дані, машинне навчання, аналітика	Compute Engine, BigQuery, Cloud Run, Vertex AI	300 USD на 90 днів. cloud.google.com/free

Для навчальних проєктів і першого знайомства рекомендується AWS - він має найбільшу кількість навчальних матеріалів, найбільшу спільноту і є найбільш затребуваним на ринку праці в Україні та Польщі. Azure є природним вибором для роботи з корпоративними замовниками Microsoft-стеку. GCP цікавий насамперед для роботи з великими масивами даних та ML. Важливо: усі три провайдери надають безкоштовні рівні, що дозволяють студентам розпочати практику без фінансових витрат.

Моделі розгортання хмари

Окрім питання «хто надає послугу», важливо розуміти «де і як розгортається хмара». Існує три основні моделі розгортання (deployment models), і вибір між ними визначається вимогами до безпеки, регуляторних зобов'язань, бюджету і рівня контролю.

Публічна хмара (Public Cloud)

Ресурси провайдера (AWS / Azure / GCP) доступні через Інтернет усім клієнтам. Дані різних клієнтів повністю ізольовані одне від одного - завдяки технологіям мультиоренди і шифруванню, але фізично знаходяться на одному і тому ж обладнанні. Це найпоширеніша і найдоступніша модель.

Переваги: найнижча вартість (нуль капітальних витрат), швидкий старт, автоматичне масштабування, підтримка провайдера, глобальна доступність. Недоліки: менший контроль над фізичним розташуванням даних, можливі обмеження для регульованих галузей.

Типові користувачі: стартапи, малий та середній бізнес, розробники для навчальних і особистих проєктів. Приклад: веб-сайт університету, розгорнутий на AWS EC2, або телеграм-бот, що працює на Google Cloud Run.

Приватна хмара (Private Cloud)

Хмарна інфраструктура виключно для однієї організації. Може розміщуватись у власному серверному залі компанії або у орендованому приміщенні (co-location - послуга розміщення власного обладнання у ЦОД провайдера). Ключова відмінність від «просто власних серверів» - наявність хмарних технологій управління: самообслуговування, еластичність, автоматизація, - але у приватному середовищі.

Популярні платформи для побудови приватної хмари: OpenStack (відкрита хмарна платформа, яку підтримує велика спільнота), VMware vSphere + vCloud Director, Microsoft Azure Stack (розширення Azure для розгортання у власному ЦОД).

Переваги: повний контроль, максимальна безпека, відповідність регуляторним вимогам. Недоліки: висока вартість (фактично - як власні сервери плюс ліцензії на хмарне програмне забезпечення), потреба у власній команді адміністраторів. Типові користувачі: банки, державні органи, оборонна галузь, медичні установи.

Гібридна хмара (Hybrid Cloud)

Поєднання приватної та публічної хмари з налагодженою інтеграцією між ними. Критично важливі або регульовані дані зберігаються у приватній частині, все інше - у публічній. Ця модель складніша в управлінні, але дає максимальну гнучкість.

Детальний приклад: великий український банк розробляє мобільний застосунок для клієнтів. Вимоги НБУ (Національного банку України - центрального банку та регулятора банківської системи) зобов'язують зберігати персональні фінансові дані клієнтів виключно на серверах, розташованих в Україні. При цьому банк хоче використати переваги хмари для решти функцій. Рішення - гібридна архітектура:

- ✓ Бекенд (backend - серверна частина застосунку) з фінансовими транзакціями та даними клієнтів → приватна хмара на OpenStack у власному ЦОД банку в Києві.
- ✓ Публічний сайт, маркетинговий контент, статичні файли → AWS CloudFront (CDN - Content Delivery Network - Мережа Доставки Контенту: розподілена мережа серверів по всьому світу для швидкої доставки контенту).
- ✓ Аналітика і ML-моделі на анонімізованих (знеособлених) даних → GCP BigQuery.

Результат: банк виконує вимоги НБУ щодо локалізації даних і водночас використовує глобальну хмарну інфраструктуру для публічних сервісів і аналітики. Це і є гібридна хмара на практиці.

Модель	Де знаходиться інфраструктура	Переваги	Недоліки	Типовий користувач
Публічна хмара (Public Cloud)	ЦОД провайдера (AWS / Azure / GCP) у різних країнах	Нуль капітальних витрат, швидкий старт, автоматичне масштабування, глобальна доступність	Менший контроль, дані на серверах провайдера	Стартапи, малий і середній бізнес, розробники, студенти

Модель	Де знаходиться інфраструктура	Переваги	Недоліки	Типовий користувач
Приватна хмара (Private Cloud)	Власний ЦОД організації або орендоване приміщення	Повний контроль, максимальна безпека, відповідність регуляторним вимогам	Висока вартість, потрібна власна команда адміністраторів, складне масштабування	Банки, державні органи, медичні установи, оборонна галузь
Гібридна хмара (Hybrid Cloud)	Частина - у приватному ЦОД, частина - у публічній хмарі; обидві частини інтегровані	Безпека для критичних даних + масштабованість і гнучкість публічної хмари	Складніше управління, потрібна налагоджена інтеграція між двома середовищами	Великі корпорації, банки з публічним сайтом, медіакомпанії, телеком

ПІДСУМОК ЛЕКЦІЇ

Ця лекція є вступною до дисципліни «Розгортання інформаційних систем» і закладає концептуальний фундамент для всіх наступних тем. Підсумуємо ключові положення:

1. **Розгортання (deployment)** - це процес перенесення готового програмного забезпечення з середовища розробки у виробниче середовище, де воно стає доступним для реальних користувачів. Це не просто «завантажити файли», а комплексний інженерний процес, що включає підготовку інфраструктури, налаштування безпеки, мережі і моніторингу.

2. **Розгортання стало хмарним** через три фундаментальні переваги: швидкість (хвилини замість тижнів), масштабованість (автоматична, без купівлі нового заліза) і економічна модель (OpEx - оплата за факт, а не CapEx - велика сума одразу).

3. **Еволюція пройшла три епохи:** фізичні сервери (10–20 % ефективність, тижні на розгортання) → віртуалізація (60–80 % ефективність, хвилини) → хмарні обчислення (90 %+ ефективність на рівні провайдера, секунди/хвилини, оплата за факт).

4. **П'ять обов'язкових характеристик хмари за NIST:** самообслуговування на вимогу, широкий мережевий доступ, пул ресурсів, швидка еластичність, вимірювана послуга. Сервіс, що не має хоча б однієї з них, не є «справжньою хмарою».

5. **Три провідні провайдери** - AWS (~32 %), Azure (~23 %), GCP (~10 %) - усі надають безкоштовні рівні для студентів. AWS - найбільш затребуваний на ринку праці в Україні.

6. **Три моделі розгортання** - публічна, приватна, гібридна хмара - обираються залежно від вимог до безпеки, регуляторних зобов'язань і бюджету.

КОНТРОЛЬНІ ПИТАННЯ

1. Дайте визначення поняття «розгортання інформаційних систем». Що входить до цього процесу і чому розгортання не зводиться до простого «завантаження файлів на сервер»?
2. Поясніть місце розгортання у SDLC (Software Development Life Cycle - Життєвий цикл розробки програмного забезпечення). Яких фахівців залучають до цього етапу?
3. Що таке DevOps (Development + Operations - Розробка + Операції)? Яку проблему вирішила ця концепція і чому вона виникла?
4. Поясніть, чому сучасне розгортання інформаційних систем нерозривно пов'язане з хмарними обчисленнями. Назвіть три ключові причини переходу на хмару.
5. Що таке CapEx (Capital Expenditure - капітальні витрати) і OpEx (Operational Expenditure - операційні витрати)? Чому перехід від CapEx до OpEx є суттєвою перевагою хмарної моделі?
6. Дайте визначення хмарних обчислень за стандартом NIST SP 800-145. Поясніть кожний ключовий елемент визначення своїми словами.
7. Перелічіть і охарактеризуйте п'ять обов'язкових характеристик хмарних обчислень за NIST. Чому без хоча б однієї з них сервіс не вважається «справжньою хмарою»?
8. Опишіть три епохи еволюції розгортання ІС (інформаційних систем). Яку ключову проблему вирішила кожна наступна епоха? Наведіть конкретні числові показники.
9. Поясніть різницю між VM (Virtual Machine - Віртуальна Машина) і контейнером (container). Які технології є найпопулярнішими у кожній категорії?
10. Чим публічна хмара відрізняється від приватної? Наведіть по одному реальному прикладу, де кожна модель є оптимальним вибором, і обґрунтуйте.
11. Поясніть, що таке гібридна хмара і в яких випадках її обирають. Чому великі банки в Україні часто використовують саме гібридну модель? Які регуляторні причини цього?
12. Порівняйте AWS, Azure та GCP: частка ринку, ключові сервіси, на який тип завдань кожен найкраще підходить. Яким провайдером рекомендується розпочати знайомство студенту?
13. Поясніть концепцію auto-scaling (автоматичне масштабування). Наведіть два приклади з реального бізнесу, де ця функція дає суттєву економічну перевагу.
14. Що таке CDN (Content Delivery Network - Мережа Доставки Контенту)? Як ця технологія пов'язана з хмарними обчисленнями і яку проблему вирішує?

Інтернет-джерела та навчальні платформи

1. AWS Free Tier - безкоштовний рівень для студентів (12 місяців). - Режим доступу:
<https://aws.amazon.com/free/>
2. Azure for Students - 100 USD кредит без кредитної картки. - Режим доступу:
<https://azure.microsoft.com/en-us/free/students/>
3. Google Cloud Free Program - 300 USD на 90 днів. - Режим доступу:
<https://cloud.google.com/free>
4. AWS Training and Certification - безкоштовні навчальні курси AWS. - Режим доступу: <https://aws.amazon.com/training/>
5. Microsoft Learn - безкоштовне навчання Azure. - Режим доступу:
<https://learn.microsoft.com/>
6. Google Cloud Skills Boost - навчальна платформа GCP. - Режим доступу:
<https://www.cloudskillsboost.google/>
7. Gartner: Cloud Computing Market Analysis 2024. - Режим доступу:
<https://www.gartner.com/en/information-technology/insights/cloud-strategy>
8. CNCF (Cloud Native Computing Foundation - Фонд хмарних обчислень) - стандарти контейнеризації та оркестрації. - Режим доступу: <https://www.cncf.io/>

ЛЕКЦІЯ 2. Моделі хмарних сервісів: iaas, paas та saas - що обрати і коли

Мета лекції

Сформувати у студентів чітке розуміння трьох ключових моделей хмарних сервісів - IaaS (Infrastructure as a Service - Інфраструктура як послуга), PaaS (Platform as a Service - Платформа як послуга) та SaaS (Software as a Service - Програмне забезпечення як послуга). Пояснити, чим кожна модель відрізняється від інших, хто за що відповідає у кожному випадку, і - найголовніше - навчити студентів обирати правильну модель для конкретного завдання. Закласти практичну основу для самостійного розгортання (деплою) першого навчального проекту у хмарі.

План лекції

1. Що таке моделі хмарних сервісів і навіщо їх розрізняти.
2. IaaS (Інфраструктура як послуга) - «орендуємо порожній офіс».
3. PaaS (Платформа як послуга) - «орендуємо офіс із меблями».
4. SaaS (Програмне забезпечення як послуга) - «просто користуємось».
5. Порівняння моделей: «пірамідка» контролю та відповідальності.
6. Як обрати модель для свого проекту.

ЩО ТАКЕ МОДЕЛІ ХМАРНИХ СЕРВІСІВ І НАВІЩО ЇХ РОЗРІЗНЯТИ

Ключова ідея: хто за що відповідає

Уявіть, що вам потрібно відкрити невеликий ресторан. Ви можете піти трьома різними шляхами. Перший шлях - побудувати власну будівлю з нуля: самостійно зробити фундамент, стіни, підвести електрику та водопостачання, а потім облаштувати кухню та зал. Максимум контролю - але скільки часу і грошей! Другий шлях - орендувати готове приміщення з кухнею та меблями: прийшли, розклали посуд, найняли кухаря - і ресторан відкрито. Третій шлях - замовляти готову їжу з доставкою: жодної кухні, жодного персоналу, просто телефонуєте і отримуєте страви.

З хмарними сервісами - рівно та сама логіка. IaaS - це перший варіант: отримуєте «порожнє приміщення» (сервер у хмарі) і налаштовуєте все самостійно. PaaS - другий варіант: отримуєте готове середовище, де залишається тільки написати код. SaaS - третій: просто відкриваєте браузер і отримуєте готовий продукт.

Ключове питання, яке розрізняє ці три моделі - «хто за що відповідає?». У кожній моделі є чіткий розподіл: щось робить провайдер (AWS, Google, Microsoft), а щось - ви як розробник або адміністратор. Чим вищий рівень моделі - тим більше бере на себе провайдер і тим менше залишається вам.

1.2. Головна «пірамідка», яку треба запам'ятати

Перш ніж заглиблюватися в деталі кожної моделі, засвойте один простий принцип, який допоможе вам завжди орієнтуватися:

- IaaS - сам налаштовуєш усе, крім заліза.
- PaaS - налаштовуєш тільки свій код і дані.
- SaaS - просто відкрив і користуєшся.

Чим вище по моделі - тим менше морочишся з інфраструктурою, але й менше контролю маєш. Це і є та «пірамідка» відповідальності: внизу - IaaS з максимумом контролю, вгорі - SaaS з мінімумом турбот. Запам'ятайте цей принцип - він допоможе вам на будь-якому технічному інтерв'ю та в роботі.

Чому це важливо знати студенту першого курсу

На перший погляд може здатися, що ці три аббревіатури - лише теорія. Але насправді розуміння різниці між IaaS, PaaS і SaaS - це практична навичка, яка знадобиться вам вже під час навчання.

По-перше, при виконанні лабораторних та дипломної роботи ви самостійно обиратимете, де і як розгортати свій застосунок. Неправильний вибір - зайві витрати часу або гроші. По-друге, на технічних інтерв'ю питання «У чому різниця між IaaS та PaaS?» є стандартним для junior-позицій (позицій початкового рівня) у будь-якій IT-компанії, що використовує хмару - а це майже всі компанії сьогодні. По-третє, правильний вибір моделі безпосередньо впливає на вартість вашого проекту. Наприклад, запустити навчальний бот або сайт на PaaS-платформі Railway - безкоштовно. Те саме на IaaS з неправильними налаштуваннями може коштувати вам \$20–50 на місяць.

Характеристика	IaaS (Інфраструктура як послуга)	PaaS (Платформа як послуга)	SaaS (Програмне забезпечення як послуга)
Що отримуєте	Віртуальні сервери, сховище, мережу - «сире» залізо у хмарі	Готове середовище для розробки: сервер + ОС + бібліотеки + база даних	Готовий продукт. Просто відкрив браузер і користуєшся
Що робите самостійно	Встановлюєте ОС (операційну систему), налаштовуєте веб-сервер, базу даних, безпеку - все	Пишете код і завантажуєте його. Решта - турбота провайдера	Нічого технічного. Тільки вводите дані та працюєте з інтерфейсом
За що платить провайдер (відповідає)	Тільки за фізичне обладнання та мережу датацентру	За обладнання + операційну систему	За все: обладнання, ОС, код, безпеку,

Характеристика	IaaS (Інфраструктура як послуга)	PaaS (Платформа як послуга)	SaaS (Програмне забезпечення як послуга)
Контроль розробника	Максимальний: можна налаштувати буквально все	Середній: контролюєш тільки свій код і дані	Мінімальний: використовуєш те, що дає провайдер
Приклади	AWS EC2, Google Compute Engine, Microsoft Azure Virtual Machines	Heroku, Google App Engine, Railway, Vercel, Render	Gmail, Google Docs, Notion, Slack, Figma, GitHub
Для кого підходить	DevOps-інженери, архітектори систем, великі компанії з особливими вимогами	Розробники, стартапи, студентські проекти - коли треба швидко	Будь-який користувач, включно з нетехнічними спеціалістами

IaaS - ІНФРАСТРУКТУРА ЯК ПОСЛУГА

Що таке IaaS - деталі

IaaS (Infrastructure as a Service - Інфраструктура як послуга) - це модель, при якій хмарний провайдер надає вам віртуальні обчислювальні ресурси: сервери (VM - Virtual Machines, віртуальні машини), сховища даних та мережеве обладнання. Ви отримуєте доступ до цих ресурсів через Інтернет і платите лише за те, що фактично використовуєте.

Важливо зрозуміти суть слова «інфраструктура». Це - найнижчий рівень ІТ-системи: фізичне обладнання, мережа, сховище. При IaaS провайдер управляє саме цим рівнем - фізичним залізом у своєму датацентрі. Все, що знаходиться «вище» - операційна система, веб-сервер, база даних, ваш застосунок - ваша відповідальність.

Практично це виглядає так: ви заходите у веб-консоль AWS (наприклад, за адресою console.aws.amazon.com), натискаєте кілька кнопок або виконуєте команди в терміналі - і за 30–60 секунд у вас є новий сервер з будь-якою кількістю процесорних ядер, оперативної пам'яті та простору на диску. Далі - ваша зона відповідальності.

Розподіл відповідальності в IaaS

Провайдер (AWS / Azure / GCP) відповідає за:	Ваша відповідальність:
Фізичні сервери та обладнання датацентру	Операційна система (Ubuntu, CentOS, Windows Server тощо)

Провайдер (AWS / Azure / GCP) відповідає за:	Ваша відповідальність:
Мережеве обладнання, кабелі, комутатори	Веб-сервер (Nginx або Apache)
Безперебійне електроживлення та охолодження	База даних (MySQL, PostgreSQL, MongoDB)
Фізична безпека датацентру	Ваш застосунок та його залежності (бібліотеки)
Гіпервізор (програма, що управляє VM)	Налаштування брандмауера (firewall - мережевого фільтра) та безпеки
Доступність мережі до вашого сервера	Резервне копіювання даних

Ця таблиця дуже важлива - її треба засвоїти. Провайдер дає вам «порожній сервер» у мережі. Далі ви, як системний адміністратор або DevOps-інженер (інженер з розробки та операцій), налаштовуєте все необхідне для роботи вашого застосунку.

Детальний приклад: розгортання веб-застосунку на AWS EC2

Розглянемо покрокову процедуру, щоб зрозуміти, що реально означає IaaS на практиці. Завдання: розгорнути простий Python Flask-застосунок (мінімалістичний веб-фреймворк) на AWS EC2 (Amazon Web Services Elastic Compute Cloud - Еластична хмарна обчислювальна служба Amazon).

Крок 1. Запуск інстансу (екземпляра) VM. Заходимо у консоль AWS і запускаємо інстанс типу t2.micro (найменший безкоштовний сервер: 1 vCPU (virtual Central Processing Unit - віртуальний процесор) та 1 GB RAM (Random Access Memory - оперативна пам'ять)). Обираємо операційну систему Ubuntu 24.04 LTS (Long Term Support - версія з довгостроковою підтримкою). Процедура займає 30–60 секунд.

Крок 2. Підключення через SSH. SSH (Secure Shell - протокол безпечного з'єднання) - це інструмент, що дозволяє підключитися до сервера через термінал та керувати ним командним рядком. Виконуємо у терміналі: `ssh -i my-key.pem ubuntu@<IP-адреса-сервера>`. Ми «всередині» нашого сервера у хмарі.

Крок 3. Встановлення необхідного ПЗ. Виконуємо команди: `sudo apt update && sudo apt install python3 python3-pip nginx` - встановлюємо Python (мову програмування), pip (менеджер пакетів Python) та Nginx (веб-сервер). Потім: `pip install flask` - встановлюємо Flask.

Крок 4. Завантаження коду. Копіюємо код нашого застосунку на сервер: `git clone https://github.com/user/myapp.git` або за допомогою SCP (Secure Copy Protocol - протокол безпечного копіювання файлів).

Крок 5. Налаштування Nginx та запуск. Налаштовуємо Nginx як reverse проху (зворотний проксі-сервер - посередник між користувачем і застосунком), щоб запити з порту 80 (HTTP) перенаправлялись до нашого Flask-застосунку. Запускаємо застосунок.

Крок 6. Налаштування безпеки. Налаштовуємо Security Groups (групи безпеки в AWS - правила для мережевого трафіку): відкриваємо порт 80 (HTTP) та 443 (HTTPS - захищений HTTP). Решту портів закриваємо.

Весь цей процес займає у досвідченого розробника 30–60 хвилин. Початківця - кілька годин. Але він дає повне розуміння того, як «під капотом» працює будь-який веб-сервіс в Інтернеті. Саме тому важливо пройти цей шлях хоча б раз - навіть якщо надалі ви будете користуватися PaaS.

Основні провайдери IaaS та безкоштовні ресурси

Критерій	AWS EC2 (Amazon Web Services Elastic Compute Cloud)	Azure Virtual Machines (Microsoft Azure)	Google Compute Engine (GCP - Google Cloud Platform)
Частка ринку (2024)	~32% - лідер ринку	~23% - стабільне друге місце	~12% - третє місце, активно зростає
Безкоштовний рівень (Free Tier)	t2.micro: 750 годин/міс протягом 12 місяців після реєстрації	B1S: 750 годин/міс протягом 12 місяців	e2-micro: 1 безкоштовний інстанс постійно (без обмежень часу!)
Сильні сторони	Найбільша екосистема: понад 200 сервісів. Найбільша кількість навчальних матеріалів	Ідеальна інтеграція з продуктами Microsoft: Windows, Active Directory, Office 365	Сильна аналітика та AI (штучний інтелект): BigQuery, Vertex AI
Де вчитися безкоштовно	AWS Skill Builder: skillbuilder.aws	Microsoft Learn: learn.microsoft.com	Google Cloud Skills Boost: cloudskillsboost.google

Рекомендація для студентів: зареєструйтесь на AWS Free Tier (aws.amazon.com/free) - отримаєте 12 місяців безкоштовного доступу до EC2 t2.micro, S3 (Simple Storage Service - хмарне сховище файлів), RDS (Relational Database Service - хмарна реляційна база даних) та десятків інших сервісів. Цього більш ніж достатньо для всіх навчальних завдань курсу.

Коли IaaS - правильний вибір

IaaS вибирають тоді, коли стандартних можливостей PaaS недостатньо і потрібен тонкий контроль над конфігурацією. Ось типові сценарії:

- Нестандартна конфігурація ОС (операційної системи): наприклад, потрібна конкретна версія ядра Linux або спеціальні драйвери для роботи з апаратним забезпеченням.
- Висока продуктивність: GPU (Graphic Processing Unit - графічний процесор) для навчання ML (Machine Learning - машинного навчання) моделей або рендеру відео.
- Регуляторні вимоги: деякі галузі (банківська, медична) вимагають повного контролю над середовищем для відповідності стандартам безпеки ISO 27001, PCI DSS (Payment Card Industry Data Security Standard - стандарт безпеки даних індустрії платіжних карток).
- Специфічне мережеве налаштування: складні VPN (Virtual Private Network - віртуальна приватна мережа), балансування навантаження з нестандартними правилами.

РaaS - ПЛАТФОРМА ЯК ПОСЛУГА

Що таке РaaS - деталі

РaaS (Platform as a Service - Платформа як послуга) - це модель, при якій провайдер надає вам не просто сервер, а повністю готове середовище для розробки та запуску застосунків. Ви отримуєте вже налаштовану операційну систему, веб-сервер, систему управління базами даних, засоби розгортання - і просто завантажуєте свій код.

РaaS буквально приховує від вас всю складність інфраструктури. Ви більше не думаєте, яка версія Ubuntu встановлена на сервері, як налаштований Nginx або де зберігаються логи (журнали подій) системи. Ваше єдине завдання - написати якісний код і «залити» (задеплоїти) його на платформу.

Пригадаємо аналогію з рестораном: РaaS - це коворкінг-простір з повністю обладнаним робочим місцем. Стіл, стілець, швидкий інтернет, кава і навіть принтер - все вже є. Ви просто прийшли зі своїм ноутбуком і почали працювати. Нікого не хвилює, яка у вас модель крісла або як налаштовано маршрутизатор.

Як виглядає деплой на РaaS - практичний приклад

Порівняймо той самий Flask-застосунок, але тепер розгортаємо його через Railway (railway.app) - одну з найпростіших РaaS-платформ.

Крок 1. Реєстрація. Заходимо на railway.app, реєструємось через GitHub (безкоштовно).

Крок 2. Підключення репозиторію. Натискаємо «New Project» → «Deploy from GitHub repo» і обираємо репозиторій (сховище коду на GitHub) нашого Flask-застосунку.

Крок 3. Railway сам усе визначає. Платформа автоматично аналізує ваш код: бачить файл requirements.txt, розуміє, що це Python-проект, сама встановлює потрібні бібліотеки, налаштовує веб-сервер і запускає застосунок.

Крок 4. Готово. Через 2–3 хвилини застосунок доступний за автоматично згенерованою адресою виду myapp-production.up.railway.app. Без SSH, без Nginx, без команд в терміналі.

Порівняйте: IaaS - 30–60 хвилин і потрібні знання адміністрування. PaaS - 3–5 хвилин і потрібне лише вміння писати код. Саме тому PaaS став стандартом для стартапів і навчальних проектів.

Основні PaaS-платформи для студентів

Платформа	Мови / стек	Безкоштовний план	Коли використовувати
Heroku	Python, Node.js, Ruby, Java, Go, PHP	Eco dynos: 1000 год/міс (сплять після 30 хв неактивності)	Перші pet-проекти (особисті проекти для портфолію), вивчення деплою
Railway	Python, Node.js, Go, Ruby, Java, Docker	\$5 кредитів/міс безкоштовно - вистачає для невеликих проектів	Студентські та дипломні проекти - найпростіший деплой (розгортання)
Render	Python, Node.js, Ruby, Go, Rust, Docker	Безкоштовний план для веб-сервісів (сплять після неактивності)	Альтернатива Heroku, зручний безкоштовний PostgreSQL
Vercel	Next.js, React, Vue, SvelteKit, Python (API)	Безкоштовний Hobby-план з необмеженими деплоями	Фронтенд-проекти та API. Найкраща інтеграція з GitHub
Google App Engine	Python, Java, Node.js, Go, PHP, Ruby	Безкоштовний рівень F1: 28 год/день безкоштовно	Якщо вже користуєтесь Google Cloud - натуральний вибір

Рекомендація: для більшості завдань цього курсу використовуйте Railway або Render - вони найпростіші у роботі, мають безкоштовні плани і чудову документацію українською та англійською. Посилання: railway.app та render.com.

Обмеження PaaS - що треба знати

PaaS - це зручно, але не всесильно. Є кілька важливих обмежень, про які варто пам'ятати:

- Менше гнучкості: ви не можете встановити довільне системне програмне забезпечення або змінити конфігурацію ОС. Все, що «нижче» вашого коду - поза вашим контролем.
- Прив'язка до платформи (vendor lock-in): застосунок, оптимізований під Heroku, може вимагати доопрацювань для переходу на Railway або AWS. Це не критично, але треба враховувати.
- Безкоштовні плани мають обмеження: більшість безкоштовних PaaS-сервісів «засинають» після 30 хвилин неактивності. Перший запит після «сну» займає 5–20 секунд. Для навчальних проєктів - нормально. Для реального продукту - потрібен платний план.
- Вища ціна при масштабуванні: при великому трафіку PaaS стає дорожчим за IaaS, оскільки ви платите за зручність.

SaaS - ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ЯК ПОСЛУГА

Що таке SaaS - деталі

SaaS (Software as a Service - Програмне забезпечення як послуга) - це найвищий рівень абстракції серед трьох моделей. При SaaS провайдер надає вам повністю готовий, розгорнутий і підтримуваний програмний продукт, доступний через Інтернет. Ви не думаєте ні про сервери, ні про код, ні про бази даних - ви просто використовуєте функціонал застосунку.

Якщо ви сьогодні писали листи в Gmail, редагували файл у Google Docs або переписувались у Slack - ви вже користуєтесь SaaS. І жодного разу не замислювались, скільки серверів обробляє ваші запити або яку базу даних використовує Gmail для зберігання листів. Це і є суть SaaS: технологія повністю прихована за зручним інтерфейсом.

Повертаємось до аналогії з рестораном: SaaS - це коли ви замовляєте доставку їжі додому. Вам не потрібна кухня, не потрібен кухар, не потрібна навіть тарілка - принесуть в контейнері. Ви платите лише за результат - готову страву. Але ви повністю залежите від того, що пропонує меню.

SaaS у повсякденному житті розробника

Важливо розуміти: SaaS - це не тільки Gmail та Google Docs. Розробники щодня використовують десятки SaaS-інструментів у своїй роботі, навіть не замислюючись про це.

Категорія	Приклади SaaS-сервісів	Що робить розробник за лаштунками	Що ви як користувач бачите
Email (електронна пошта)	Gmail, Outlook, ProtonMail	Підтримує сервери, фільтри спаму, шифрування, реплікацію даних	Просто пишете і отримуєте листи у браузері
Редактори та офіс	Google Docs, Notion, Confluence	Управляє версіями документів, синхронізацією в реальному часі	Відкрили посилання, почали писати - все зберігається автоматично
Комунікація та командна робота	Slack, Discord, Zoom, Microsoft Teams	Підтримує P2P і серверні з'єднання, запис відео, сховище файлів	Натиснули «Приєднатися» - відразу в ефірі
Дизайн та прототипування	Figma, Canva	Рендеринг (відображення) векторної графіки у браузері в реальному часі	Малюєте інтерфейси разом із командою одночасно
Розробка та хостинг коду	GitHub, GitLab, Bitbucket	Зберігає мільярди рядків коду, обробляє git push/pull від мільйонів розробників	Завантажили код командою git push - він уже доступний онлайн

Зверніть увагу: GitHub - це теж SaaS. Коли ви виконуєте git push (команда відправки змін у систему контролю версій Git), ваш код зберігається на серверах GitHub (які знаходяться в Azure - хмарній платформі Microsoft). Ви ніколи не думаєте про ці сервери - просто користуєтесь сервісом.

SaaS з точки зору розробника: будуємо свій SaaS

Тут важливий момент, який часто упускають: як студент спеціальності «Комп'ютерні науки», ви будете не тільки користуватись SaaS, але й будувати його. Адже більшість сучасних стартапів - це SaaS-продукти.

Що означає «побудувати SaaS»? Це - розробити веб-застосунок, який надає якийсь сервіс через Інтернет за підписку або оплату. Наприклад, сервіс для автоматичної перевірки коду на помилки, платформа для онлайн-опитувань, інструмент для планування завдань команди. При цьому ви самі обираєте, на чому будувати свій SaaS: можна розгорнути на IaaS (повний контроль) або на PaaS (швидкість розробки).

Реальний приклад для розуміння масштабу: Notion (популярний SaaS-інструмент для нотаток та проектного менеджменту, notion.so) розпочинався як застосунок двох розробників,

розгорнутий на AWS. Сьогодні він обслуговує понад 30 мільйонів користувачів, але для кінцевого користувача нічого не змінилося - просто відкрив browser і почав писати нотатки.

ПОРІВНЯННЯ МОДЕЛЕЙ: «ПІРАМІДКА» КОНТРОЛЮ ТА ВІДПОВІДАЛЬНОСТІ

Головна «пірамідка» - поглиблений розбір

Повернемося до головного принципу, який ми вже згадали на початку, але тепер - з повним розумінням кожного рівня. Запам'ятайте цей принцип назавжди:

- IaaS - сам налаштовуєш усе, крім заліза.
- PaaS - налаштовуєш тільки свій код і дані.
- SaaS - просто відкрив і користуєшся.

Чим вище по моделі - тим менше морочишся з інфраструктурою, але й менше контролю маєш.

Тепер розберемо детально. При IaaS ви контролюєте: операційну систему та її версію, веб-сервер та його конфігурацію, базу даних і її налаштування, безпеку на рівні мережі та ОС, версії всіх встановлених бібліотек. При PaaS ви контролюєте: код вашого застосунку, структуру бази даних (схему), змінні середовища (environment variables) - паролі та конфігурацію. При SaaS ви контролюєте: лише свої дані (тексти, файли, налаштування профілю) в межах інтерфейсу.

Аналогія, яка все пояснює: від будівництва до готелю

Ось ще одна аналогія, яка допоможе остаточно закріпити різницю між моделями. Уявіть, що вам потрібне місце для роботи.

IaaS - купуєте земельну ділянку і будуєте офіс самостійно. Ви обираєте кожен цеглу, кожний кабель, кожну розетку. Повний контроль - але потрібні знання архітектора, будівельника та електрика. Зате офіс буде рівно таким, яким ви хочете.

PaaS - орендуєте готовий офіс з меблями та інтернетом. Прийшли з ноутбуком - і відразу працюєте. Не можете зламати стіну і поставити там сервер, але для 95% задач цього і не треба. Договір оренди - щомісячна оплата.

SaaS - живете в готелі і замовляєте послуги. Кімната прибирається, білизна міняється, є рум-сервіс. Але змінити планування кімнати або встановити власний замок на двері - неможливо. Ви гість, а не власник.

Підсумкова таблиця порівняння

Модель	Одне речення - суть	Аналогія з реального життя	Запам'ятати
IaaS (Infrastructure as a Service - Інфраструктура як послуга)	Орендуєш порожній офіс - меблі, ремонт, охорону організовуєш сам	Оренда порожнього приміщення під офіс - є стіни, є електрика, далі сам	Максимум роботи, максимум свободи
PaaS (Platform as a Service - Платформа як послуга)	Орендуєш офіс із меблями - приходиш і одразу працюєш	Коворкінг: стіл, інтернет, кава вже є - приніс ноутбук і починай	Середня робота, середня свобода
SaaS (Software as a Service - Програмне забезпечення як послуга)	Користуєшся послугами аутсорс-команди - платиш за результат, не за офіс	Таксі: не купуєш машину, не наймаєш водія - просто замовив і поїхав	Мінімум роботи, мінімум свободи

ЯК ОБРАТИ МОДЕЛЬ ДЛЯ СВОГО ПРОЕКТУ

Практична таблиця вибору

Ситуація	IaaS	PaaS	SaaS
Студентський / дипломний проект	Можна, але зайве - доведеться налаштовувати багато речей вручну	Ідеально - Railway або Render: безкоштовно, просто, швидко	Якщо потрібен готовий інструмент (GitHub для коду, Notion для нотаток)
Стартап на ранній стадії (MVP - Minimum Viable Product, мінімально життєздатний продукт)	Зарано - витратите час на адміністрування замість продукту	Ідеально - зосереджуєтесь на коді, платите лише при зростанні	Для бізнес-інструментів: Slack для команди, Figma для дизайну
Великий production-сервіс (продуктивне середовище) зі специфічними вимогами	Ідеально - повний контроль над конфігурацією, можливість тонкого налаштування	Може не вистачити гнучкості для нестандартних конфігурацій	Не підходить - потрібен власний код, а не чужий продукт
Корпоративна система (ERP, CRM, бухгалтерія)	Якщо потрібна повна ізоляція і відповідність нормативам (GDPR, ISO)	Можливо, якщо є готовий PaaS з потрібними сертифікатами	Часто ідеально: SAP, Salesforce, 1C у хмарі - зрілі рішення

Ситуація	IaaS	PaaS	SaaS
Навчальні проекти на курсі	AWS EC2 Free Tier - обов'язково спробувати хоча б раз для розуміння	Railway / Render - для більшості завдань курсу	GitHub - для зберігання коду всіх проектів

Алгоритм прийняття рішення

Якщо ви не впевнені, яку модель обрати для свого проекту, пройдіть по цьому алгоритму. Спочатку поставте собі перше питання: «Чи знаю я, що таке SSH і чи готовий витратити 1–2 години на налаштування сервера?». Якщо відповідь «ні» або «зараз не хочу» - оминайте IaaS, обирайте PaaS.

Друге питання: «Чи вимагає мій проект специфічного налаштування системи, якого не надає PaaS?». Для 90% студентських проектів відповідь «ні» - значить PaaS цілком вистачить.

Третє питання: «Це мій власний застосунок чи я використовую чийсь готовий продукт?». Якщо це чийсь готовий продукт (GitHub, Notion, Figma) - це SaaS, і ви просто ним користуєтесь. Якщо це ваш власний код - потрібен або IaaS, або PaaS.

Конкретні рекомендації для студентів курсу

На основі реального досвіду викладання ось найефективніший шлях для студента першого курсу спеціальності 122 «Комп'ютерні науки»:

Перший місяць: Зареєструйтесь на Railway (railway.app) та GitHub (github.com). Задеployте свій перший Python або Node.js (середовище виконання JavaScript на сервері) застосунок через Railway. Це займе 15–30 хвилин і дасть практичне розуміння того, що таке деплой (розгортання).

Другий-третій місяць: Зареєструйтесь на AWS Free Tier (aws.amazon.com/free). Запустіть EC2 інстанс, підключіться через SSH, вручну налаштуйте веб-сервер. Це - «обряд посвячення» для будь-якого розробника. Після цього PaaS здаватиметься справжньою магією.

Для дипломного проекту: Обирайте Railway або Render для бекенду (серверної частини застосунку), Vercel - для фронтенду (клієнтської частини, що бачить користувач). Використання GitHub для коду є обов'язковим.

Важлива порада: не бійтесь помилятися і «ламати» тестові середовища. Безкоштовні tier (рівні) хмарних провайдерів існують саме для того, щоб ви вчилися без фінансових ризиків. Якщо щось пішло не так - просто видаліть інстанс і почніть заново.

Корисні ресурси для самостійного вивчення

Офіційна документація та навчальні платформи:

- AWS Free Tier та навчальні матеріали: aws.amazon.com/free та skillbuilder.aws

- Google Cloud Skills Boost (безкоштовні навчальні лабораторії):
cloudskillsboost.google
- Microsoft Learn (безкоштовне навчання Azure): learn.microsoft.com/uk-ua/azure
- Railway - документація: docs.railway.app
- Render - документація: render.com/docs
- Vercel - документація: vercel.com/docs

Корисні статті та курси:

- Офіційна документація AWS про різницю IaaS/PaaS/SaaS:
docs.aws.amazon.com/whitepapers/latest/aws-overview/types-of-cloud-computing.html
- Microsoft Azure: пояснення IaaS, PaaS, SaaS: azure.microsoft.com/uk-ua/overview/what-is-iaas
- Безкоштовний курс «Cloud Computing Basics» на Coursera від LearnQuest:
coursera.org (шукайте «cloud computing basics»)
- Відеоканал TechWorld with Nana на YouTube: доступні пояснення DevOps та хмарних технологій англійською: youtube.com/@TechWorldwithNana

ПІДСУМОК ЛЕКЦІЇ

Ця лекція заклала фундамент розуміння хмарних сервісів - без якого неможливо рухатись далі в дисципліні «Розгортання інформаційних систем». Давайте ще раз закріпимо головне.

Три моделі хмарних сервісів - IaaS, PaaS і SaaS - це три різні відповіді на одне запитання: «Скільки відповідальності за інфраструктуру беремо на себе?». IaaS дає максимальний контроль і вимагає максимальних знань. PaaS балансує між зручністю і гнучкістю - ідеальний вибір для більшості розробників і студентських проектів. SaaS - готовий продукт, яким просто користуються.

Пам'ятайте головну «пірамідку»: IaaS - сам налаштовуєш усе, крім заліза. PaaS - налаштовуєш тільки свій код і дані. SaaS - просто відкрив і користуєшся. Чим вище по моделі - тим менше морочишся з інфраструктурою, але й менше контролю маєш.

Найважливіший практичний висновок: жодна модель не є «найкращою» чи «найгіршою». Правильний вибір залежить від завдання. Для студентського проекту, який треба показати завтра, - Railway (PaaS) ідеальний. Для великого корпоративного сервісу з особливими вимогами безпеки - AWS EC2 (IaaS). Для спільної роботи над документами в команді - Google Docs (SaaS). Уміння обрати правильний інструмент під задачу - це і є ознака справжнього інженера.

КОНТРОЛЬНІ ПИТАННЯ

Загальне розуміння моделей

1. Що таке моделі хмарних сервісів і в чому полягає ключова різниця між IaaS, PaaS та SaaS?
2. Поясніть принцип «пірамідки» контролю та відповідальності. Як змінюється рівень контролю розробника при переході від IaaS до SaaS?
3. Яке ключове питання допомагає розрізнити три моделі хмарних сервісів?

IaaS

4. Що входить до зони відповідальності провайдера при IaaS, а що — до зони відповідальності розробника?
5. Опишіть основні кроки розгортання веб-застосунку на AWS EC2. Скільки часу займає цей процес у досвідченого розробника?
6. У яких випадках доцільно обирати IaaS? Наведіть не менше трьох сценаріїв.
7. Порівняйте безкоштовні рівні (Free Tier) трьох основних IaaS-провайдерів: AWS, Azure та Google Cloud.

PaaS

8. Чим PaaS відрізняється від IaaS з точки зору процесу розгортання застосунку?
9. Порівняйте розгортання Flask-застосунку через IaaS (AWS EC2) та PaaS (Railway). У чому головна перевага PaaS?
10. Які основні обмеження PaaS-платформ і в яких ситуаціях вони стають критичними?
11. Назвіть три PaaS-платформи, рекомендовані для студентських проектів, та їхні особливості.

SaaS

12. Наведіть п'ять прикладів SaaS-сервісів, якими розробники користуються щодня. Поясніть, що саме «приховано» за кожним з них.
13. Чому GitHub вважається SaaS-сервісом? На якій хмарній платформі він функціонує?
14. Що означає «побудувати власний SaaS»? Які моделі (IaaS або PaaS) можна використовувати для його розгортання?

Вибір моделі

15. За яким алгоритмом рекомендується обирати модель хмарного сервісу для власного проекту?
16. Яку модель і яку платформу рекомендується використовувати для дипломного проекту і чому?
17. Поясніть поняття «vendor lock-in» та з якою моделлю хмарних сервісів воно найбільше пов'язане.

ЛЕКЦІЯ 3. Архітектура та принципи хмарних додатків: Serverless-архітектура та принципи хмарних додатків

У Лекції 2 ми з'ясували, де запускати код - IaaS, PaaS або SaaS. Тепер розберемося, як проектувати код так, щоб він повністю використовував переваги хмари.

Три ключові теми цієї лекції:

- Serverless - архітектура, де ви платите тільки за реальне виконання коду.
- Cloud-Native і мікросервіси - підхід, при якому додаток будується з урахуванням можливостей хмари.
- 12-Factor App - 12 практичних принципів, які роблять хмарний додаток якісним.

Мета лекції

Зрозуміти принципи Serverless та FaaS, навчитися будувати Cloud-Native додатки і дотримуватися 12 принципів якісного хмарного проекту.

Що таке архітектура хмарного додатку

Архітектура програмного забезпечення - це фундаментальна організація системи, що визначає структуру компонентів, їх взаємозв'язки та принципи проектування і розвитку. Якщо провести аналогію з будівництвом: архітектура - це не просто зовнішній вигляд будівлі, це сукупність рішень про те, які матеріали використовувати, як розподілити навантаження, як влаштувати каналізацію, вентиляцію та електропроводку. Неправильні рішення на рівні архітектури можуть призвести до обвалу всієї конструкції, навіть якщо кожен окремий елемент виглядає бездоганно.

Хмарна архітектура (Cloud Architecture) - це сукупність технологій, компонентів та принципів, які визначають, як програмна система використовує ресурси хмарних обчислень. Вона включає рішення про те, яким чином компоненти системи взаємодіють між собою, де зберігаються дані, як система масштабується під навантаженням, яким чином забезпечується безперебійна робота та відновлення після збоїв.

Ключові питання, на які відповідає архітектура хмарного додатку:

- Де і як виконується код (сервери, контейнери, функції)?
- Як система реагує на зростання навантаження (масштабування)?
- Що відбувається при збої окремих компонентів (відмовостійкість)?
- Як відбувається обмін даними між компонентами (комунікація)?
- Де і як зберігаються дані (бази даних, сховища, кеш)?
- Як забезпечується безпека (автентифікація, авторизація, шифрування)?
- Скільки коштуватиме підтримка системи під різним навантаженням?

ПЛАН ЛЕКЦІЇ

- Serverless - без-серверна архітектура: що це і навіщо.
- FaaS (Function as a Service) та AWS Lambda.
- Cloud-Native - хмарно-орієнтований додаток, мікросервіси.
- Методологія 12-Factor App - 12 принципів якості.
- Підсумок: як обрати архітектуру для свого проекту.

SERVERLESS - ЩО ЦЕ?

Serverless (від англ. «без серверів») - це модель, в якій **ви просто пишете функцію** (шматок коду), а хмарний провайдер сам вирішує, де і коли її запустити. Вам не потрібно орендувати сервер, встановлювати Ubuntu, запускати Nginx - нічого цього!

Важливо: «Serverless» НЕ означає, що серверів немає взагалі. Фізичні сервери є - просто вони приховані від вас. Правильніша назва - «Less Server Management» (менше управління серверами).

Аналогія: Коли ви вмикаєте чайник у розетку, ви не думаєте, яка електростанція генерує струм. Ви просто отримуєте результат. Serverless - те саме: ви пишете «що треба зробити», а провайдер вирішує «як і де запустити».

Чому Serverless з'явився?

До появи Serverless кожен стартап орендував сервер і тримав його увімкненим 24 години на добу, 7 днів на тиждень. Навіть якщо вночі ніхто не заходить на сайт - рахунок іде постійно. Проблема добре видно на прикладі сервісу продажу квитків на концерти: 99% часу - повна тиша. Але в момент старту продажу квитків на популярного виконавця - тисячі запитів за кілька хвилин. Тримати потужний сервер заради 10 «пікових хвилин» на місяць - марна трата грошей. Amazon вирішили цю проблему у 2014 році, випустивши AWS Lambda - перший масовий FaaS-сервіс. Ідея проста: платиш тільки за реальний час виконання коду - в мілісекундах.

Serverless vs Традиційний сервер

Розглянемо конкретний приклад: інтернет-магазин з піком продажів у п'ятницю ввечері та мінімальним трафіком вночі у вівторок.

Критерій	Традиційний сервер	Serverless
Коли працює?	Постійно (24/7), навіть при 0 запитах	Тільки під час обробки запиту (мілісекунди)
Оплата	За весь час роботи сервера (постійно)	Тільки за фактичний час виконання функції

Масштабування	Вручну: купити/орендувати серверів треба більше	Автоматично: 10 000 запитів → 10 000 копій функції
Адміністрування	Ваша відповідальність: оновлення ОС, безпека, моніторинг	Нічого - провайдер сам відповідає за все
Cold Start	Відсутній - сервер вже запущений	Перший запит після паузи: 500–2000 мс
Вартість (1 млн запитів/міс)	EC2 t2.micro: ~\$8.5/міс	AWS Lambda: ~\$1.87/міс (у 4.5× дешевше!)

Реальний приклад - компанія iRobot (пилосос-роботи Roomba): після переходу на AWS Lambda витрати скоротилися з ~\$8 000 на місяць до ~\$1 500. Економія 81%! Причина: Serverless платить тільки за реальне використання, а не за «час очікування».

Як саме працює Serverless - крок за кроком

Розглянемо реальний приклад: обробка фото профілю, яке завантажив користувач на сайт.

- Крок 1. Користувач завантажує фото 5 MB на сайт через форму.
- Крок 2. AWS S3 зберігає оригінальний файл і автоматично генерує подію.
- Крок 3. AWS Lambda бачить цю подію і автоматично запускає вашу функцію (займає 50–2000 мс).
- Крок 4. Ваш код стискає фото до розміру 200×200 пікселів (thumbnail - мініатюра).
- Крок 5. Мініатюра зберігається назад у S3.
- Крок 6. Функція завершується, і Lambda виставляє рахунок: кілька мілісекунд × \$0.0000000167.

Ключова перевага: якщо одночасно 1000 користувачів завантажують фото - Lambda автоматично запускає 1000 копій функції. Ніяких балансувальників навантаження налаштовувати не потрібно!

FaaS - ФУНКЦІЯ ЯК ПОСЛУГА

FaaS (Function as a Service - Функція як послуга) - це конкретна реалізація Serverless. Якщо Serverless - загальна концепція «не думати про сервери», то FaaS - практичний інструмент: ви пишете одну функцію, завантажуєте її до провайдера, і він запускає її у відповідь на подію (тригер).

Кожна функція FaaS є Stateless (від англ. «без стану») - вона не «пам'ятає» попередні виклики. Кожен виклик - незалежний. Це важливо для масштабування! Детальніше про принцип Stateless - у розділі 4 (Фактор 6).

Основні FaaS-платформи

Платформа	Безкоштовний план	Особливість
AWS Lambda	1 млн запитів/міс + 400 тис. GB-сек обчислень	Найбільша екосистема, підтримка Python, Node.js, Java, Go, C#
Google Cloud Functions	2 млн викликів/міс	Проста інтеграція з BigQuery, Firebase, Google AI
Azure Functions	1 млн викликів/міс	Ідеально для .NET і Microsoft-стеку
Cloudflare Workers	100 тис. запитів/день	Найшвидший Cold Start (~5 мс!) - Edge-обчислення біля користувача
Vercel Functions	Необмежено (хобі-план)	Автодеплой з GitHub, ідеально для фронтенду

Рекомендація для початку: спробуйте AWS Lambda або Vercel Functions - найпростіший процес реєстрації та розгортання. Cloudflare Workers вибирайте, якщо критична швидкість відповіді API.

Cold Start (холодний старт) - головна проблема Serverless

Cold Start (холодний старт) - затримка при першому запуску функції після тривалої паузи. Чому це відбувається?

Після виконання функції провайдер «заморожує» контейнер (ізольоване середовище виконання) на кілька хвилин. Якщо новий запит приходить швидко - контейнер «розморожується» миттєво (Warm Start). Якщо пройшло більше 5–15 хвилин - контейнер знищується, і наступний запит запускає все з нуля.

Ситуація	Час відповіді	Що відбувається
Warm Start (теплий старт)	50–200 мс	Контейнер вже «теплий» - одразу виконує ваш код
Cold Start (холодний старт)	500–2000 мс	Контейнер завантажується з нуля: запуск середовища + завантаження коду + ініціалізація

Коли це проблема? Для REST API, де користувач чекає відповіді: затримка 2 секунди = «сайт завис». Дослідження Google: кожні 0.1 секунди затримки знижують конверсію на 1%.

Коли це не проблема? Для фонових задач: резервне копіювання, генерація звітів, нічні розсилки - 2 секунди на старті не відчутні.

Як боротися з Cold Start - 3 способи:

- Обирайте легкі мови: Node.js (~100 мс) або Python (~200 мс) замість Java (JVM потребує 500–3000 мс тільки на запуск).
- Мінімізуйте розмір пакету: менше бібліотек = швидше завантаження коду.
- Provisioned Concurrency: AWS тримає N «теплих» екземплярів функції постійно запущеними. Cold Start = 0 мс, але за це платите навіть у час простою.

Вартість Serverless - коли вигідно?

Serverless - не завжди дешевше. Все залежить від характеру навантаження.

Навантаження	AWS Lambda	EC2 t2.micro	Висновок
100 тис. запитів/міс	~\$0 (безкоштовно!)	\$8.5/міс	Lambda ідеальна - MVP, боти, навчальні проекти
1 млн запитів/міс	~\$1.87/міс	\$8.5/міс	Lambda у 4.5× дешевше - малий стартап
10 млн запитів/міс	~\$18/міс	\$8.5/міс	EC2 дешевше при рівномірному трафіку
100 млн+ запитів/міс	~\$500/міс	~\$70/міс (c5.large)	EC2 у 7× дешевше - великий production-сервіс

Правило вибору: якщо сервер завантажений більше 30% часу доби - EC2 вигідніша. Якщо є чіткі піки і тривала тиша - Serverless зекономить гроші. Для навчальних проектів Lambda безкоштовна до 1 млн запитів/міс!

Типові сценарії використання FaaS

Сценарій	Тригер (подія)	Що робить функція
Обробка зображень	Завантаження файлу в S3	Стискає фото, створює мініатюру 200×200 пікселів
Заплановані задачі (Cron)	Таймер: щодня о 02:00	Резервне копіювання БД, очищення старих записів
Webhook-обробка	HTTP POST від GitHub/Stripe/Telegram	Запускає тести, надсилає сповіщення, оновлює БД
Telegram/Viber бот	HTTP запит від Telegram API	Парсить команду, формує відповідь, надсилає через API
API для мобільного додатку	HTTP GET/POST від мобільного	Авторизація, читання/запис БД, повернення JSON

НЕ підходить: рендер відео, ML-навчання	-	Ліміт Lambda - 15 хвилин. Використовуйте EC2 або AWS Batch
---	---	--

Приклад коду AWS Lambda на Python

Ось реальна Lambda-функція. Вона приймає HTTP GET-запит і повертає привітання з іменем. Зверніть увагу на структуру: функція завжди приймає два аргументи - event (подія) та context (контекст).

```
import json
import datetime

def lambda_handler(event, context):
    # event --- вхідні дані (HTTP запит, S3 подія тощо)
    # context --- інформація про середовище виконання
    name = event.get('queryStringParameters', {}).get('name', 'Студент')
    return {
        'statusCode': 200,
        'headers': {'Content-Type': 'application/json'},
        'body': json.dumps({
            'message': f'Привіт, {name}! Це AWS Lambda.',
            'time': str(datetime.datetime.now()),
            'remaining_ms': context.get_remaining_time_in_millis()
        }, ensure_ascii=False)
    }
```

Вартість одного виклику: ~\$0.000000208 - менше тисячної частки цента. Перші 1 млн викликів на місяць - безкоштовно!

CLOUD-NATIVE - ХМАРНО-ОРІЄНТОВАНІ ДОДАТКИ

Cloud-Native (від англ. «рідний для хмари») - підхід, при якому додаток проектується з нуля з урахуванням переваг хмари: автоматичного масштабування, самовідновлення, незалежного оновлення частин системи.

Аналогія: «Lift-and-Shift» (перенести як є) - це взяти старий двоповерховий будинок і поставити його на хмарний майданчик. Він там стоїть, але не має ліфтів (автомасштабування), запасних виходів (failover), модульних кімнат (мікросервіси). Cloud-Native - це збудувати хмарочос, де кожен поверх будується і оновлюється незалежно.

Аспект	Lift-and-Shift (перенести як є)	Cloud-Native (спроектовано для хмари)
Підхід	Готовий старий додаток просто переносять на хмарний сервер	Будується з нуля або переписується з урахуванням хмари
Архітектура	Моноліт - один великий проект	Мікросервіси + контейнери (Docker) + Serverless
Виграш від хмари	Мінімальний - сервер той самий, просто орендований	Максимальний: автомасштабування, відмовостійкість, CI/CD
Приклад	Стара банківська система на AWS EC2 без змін	Netflix: 700+ мікросервісів, ~1000 деплоїв на добу

Мікросервісна архітектура

Мікросервіси (Microservices) - архітектурний підхід, при якому великий складний додаток розбивається на багато невеликих незалежних сервісів. Кожен сервіс виконує одну конкретну бізнес-функцію, має власну базу даних і розгортається незалежно від інших.

Це протилежність Монолітного додатку, де весь код - авторизація, каталог, кошик, оплата, сповіщення - знаходиться в одному проекті й розгортається як єдине ціле.

Критерій	Моноліт	Мікросервіси
Структура коду	Весь код в одному проекті	Кожна бізнес-функція - окремий сервіс у різних репозиторіях
Масштабування	Тільки весь додаток цілком (навіть непотрібні частини)	Масштабується тільки перевантажений сервіс - економія ресурсів
Деплой	Зміна 1 рядка → перебудова всього (ризик + простій)	Зміна в 1 сервісі → деплой тільки його, решта працює
Вплив збою	Впав 1 модуль → весь додаток недоступний	Впав 1 сервіс → інші продовжують працювати
Хто використовує	Стартапи, малі проекти, MVP - це нормально!	Netflix, Uber, Amazon, Spotify - після досягнення масштабу

Реальний приклад - Netflix (270 млн підписників): 700+ мікросервісів, кожен зі своєю командою. ~1000 деплоїв коду на добу без жодного планового простою. Netflix навіть використовує інструмент **Chaos Monkey** - він випадково «вбиває» сервіси в production, щоб перевірити відмовостійкість!

Порівняння архітектурних підходів

Критерій	Моноліт на IaaS	Мікросервіси	Serverless/FaaS	PaaS
Складність старту	Середня	Висока	Низька	Дуже низька
Автомасштабування	Весь додаток цілком	По кожному сервісу окремо	Повністю автоматично	Гнучке
Вартість (малий трафік)	~\$8.5/міс (EC2)	~\$20–50/міс	~\$0–2/міс	~\$5–15/міс
Підходить для	Малі проекти, MVP, 1–3 особи	Великі компанії (Netflix, Uber)	Боти, API, обробка подій	Будь-що від малого до великого

МЕТОДОЛОГІЯ 12-FACTOR APP

12-Factor App (12-факторний застосунок) - 12 принципів розробки якісних хмарних додатків, сформульованих інженерами компанії Heroku у 2011–2012 роках. Ці принципи гарантують, що ваш додаток буде легко масштабуватись, відновлюватись після збоїв і без проблем розгортатись у різних середовищах.

Методологія не прив'язана до конкретної мови програмування - вона однаково підходить для Python, Node.js, Java, Go. Використовується в Netflix, Spotify, Google, Heroku.

Важливо розуміти: ці принципи - не академічна теорія, а практичні правила, виведені з реального досвіду розгортання тисяч додатків. Кожне порушення призводить до конкретних проблем на виробничому середовищі.

Усі 12 принципів - просто і з прикладами

#	Назва	Суть	Неправильно → Правильно
1	Codebase / Єдина кодова база	Один Git-репозиторій → багато середовищ	Окремі репо для dev/prod → один GitHub-репо, деплоїться в dev/staging/prod з різними конфігами
2	Dependencies / Явні залежності	Всі бібліотеки явно задекларовані у файлі	<code>pip install flask</code> (без файлу) → <code>requirements.txt</code> або <code>package.json</code> з точними версіями
3	Config / Конфігурація у змінних середовища	Паролі та ключі - у змінних оточення, НЕ в коді	<code>API_KEY = 'abc123'</code> у коді → <code>API_KEY = os.environ['API_KEY']</code>

4	Backing Services / Зовнішні сервіси	БД, кеш, email - підключені як ресурси через URL	host=localhost жорстко в коді → DB_URL = os.environ['DATABASE_URL']
5	Build / Release / Run	Три чіткі етапи деплою - не змішувати	Ручне редагування файлів на сервері → git push → build → image → deploy автоматично
6	Processes / Stateless-процеси	Не зберігати стан у RAM між запитами	Сесії у RAM сервера → сесії у Redis, файли в S3, стан у PostgreSQL
7	Port Binding / Прив'язка до порту	Додаток сам запускає HTTP-сервер	Залежність від зовнішнього Apache → app.run(port=os.environ['PORT'])
8	Concurrency Паралельність	Масштабування через більше процесів, не потоків	1 процес з 32 потоками → 10 окремих процесів за балансувальником навантаження
9	Disposability Одноразовість	Швидкий старт (<10 сек), коректне завершення	Старт 2+ хв, SIGTERM не обробляється → старт за 3 сек, дочекатись завершення запитів
10	Dev/Prod Parity / Схожість середовищ	Dev і prod - максимально схожі	SQLite локально + PostgreSQL у prod → PostgreSQL у Docker локально = AWS RDS у prod
11	Logs / Логи	Логи - потік у stdout (стандартний вивід)	Запис у файл /var/log/app.log → print()/logging.info() у stdout → AWS CloudWatch збирає автоматично
12	Admin Processes / Адмін-процеси	Адмін-задачі - разові процеси в тому ж середовищі	Ручний скрипт на prod-сервері → python manage.py migrate (той самий Docker-образ)

Найважливіший принцип -

Фактор 1: Конфігурація у змінних середовища

Це найчастіша помилка початківців - вписувати паролі, токени та API-ключі (ключі доступу до зовнішніх сервісів) прямо в код.

Чому це небезпечно? Код рано чи пізно потрапляє у Git-репозиторій. Навіть «приватний» репозиторій може стати публічним через помилку. Боти-сканери знаходять секрети в публічних репозиторіях за лічені хвилини.

Реальний інцидент (2022 рік): Розробник випадково закомітив AWS-ключі у публічний GitHub. За 3 хвилини бот знайшов ключі та запустив 100 хмарних серверів для майнінгу криптовалюти. Рахунок від AWS за одну ніч: \$47 000 (сорок сім тисяч доларів)!

Як правильно - читаємо конфігурацію з Environment Variables (змінних оточення):

```
import os
DATABASE_URL = os.environ['DATABASE_URL'] # URL підключення до БД
API_KEY = os.environ['API_KEY']          # ключ доступу до зовнішнього API
SECRET_KEY = os.environ.get('SECRET_KEY', 'dev-only-default')
# Локально: створіть файл .env (і ОБОВ'ЯЗКОВО додайте його у .gitignore!)
# DATABASE_URL=postgresql://localhost:5432/mydb
# API_KEY=your-local-test-key
```

Фактор 2: Stateless-процеси (без стану в пам'яті)

Stateless (від англ. «без стану») означає, що додаток не зберігає нічого між запитами у своїй оперативній пам'яті (RAM). Кожен запит обробляється незалежно. Весь стан зберігається у зовнішніх сховищах: базах даних, Redis, S3.

Чому це важливо для хмарних систем?

Stateful (зі станом) - проблема: Сервер А зберігає сесію Іван у RAM. Якщо запит потрапив на сервер В - він не знає про сесію. Потрібні sticky sessions (прив'язка клієнта до конкретного сервера), що ускладнює масштабування.

Stateless (без стану) - правильно: Сесії зберігаються у Redis (зовнішньому кеш-сховищі). Будь-який з 10 серверів може обробити будь-який запит. Перезапустили сервер - користувачі нічого не помітили.

Фактор 3: Dev/Prod Parity (Схожість середовищ)

«У мене на комп'ютері все працює!» - класична фраза, яка означає, що порушено Фактор 10. Dev і prod відрізняються - і баги, які виникають тільки у prod, дуже важко знайти.

- Погана практика: SQLite локально (легка БД для розробки), PostgreSQL у prod → різна поведінка транзакцій, різні баги.
- Хороша практика: PostgreSQL у Docker-контейнері локально = AWS RDS у prod → ідентична поведінка.
- Погана практика: Python 3.9 на ноутбучі, Python 3.12 у prod → несумісні зміни API між версіями.
- Хороша практика: Однакова версія Python через pyenv або закріплена у Dockerfile.

ПІДСУМОК: ЯК ОБРАТИ АРХІТЕКТУРУ

Немає «найкращої» архітектури - є «правильна для ваших умов: розміру команди, бюджету, характеру навантаження, термінів».

Ситуація	Моноліт на IaaS	Мікросервіси	Serverless/FaaS	PaaS
Студентський/дипломний проект	Можна (EC2 Free Tier)	Зайво складно для 1 особи	Ідеально - безкоштовно!	Ідеально - Railway/Render безкоштовні
MVP стартапу	Можна	Занадто рано - зайва складність	Так - платите тільки за реальне використання	Так - швидко, зручно, без DevOps
Telegram/Viber бот	Надлишок	Зайво	Ідеально - реагує на події, платите копійки	Добре - просте розгортання
Обробка зображень/файлів	Надлишок	Зайво	Ідеально - тригер на upload, auto-scale	Можна
Рендер відео 2+ год, ML-моделі	EC2 великого розміру - правильно	ECS або Kubernetes для довгих задач	Ліміт Lambda - 15 хвилин максимум	Не вистачить часу виконання

КОРИСНІ ПОСИЛАННЯ ТА РЕСУРСИ

Офіційна документація

- AWS Lambda - документація і туторіали: docs.aws.amazon.com/lambda
- Google Cloud Functions: cloud.google.com/functions/docs
- Azure Functions: docs.microsoft.com/azure/azure-functions
- Cloudflare Workers: developers.cloudflare.com/workers

12-Factor App та Cloud-Native

- Офіційний сайт 12-Factor App (є переклад українською!): 12factor.net
- CNCF - Cloud Native Computing Foundation: cncf.io
- Патерни мікросервісів від Chris Richardson: microservices.io
- Класична стаття про мікросервіси від Мартіна Фаулера: martinfowler.com/articles/microservices.html

Практика (безкоштовно для студентів)

- AWS Free Tier - Lambda + API Gateway + S3 безкоштовно 12 місяців:
aws.amazon.com/free
- Railway - деплой Python/Node.js без налаштувань (є безкоштовний план):
railway.app
- Render - безкоштовний хостинг для веб-сервісів (альтернатива Heroku):
render.com
- Vercel Functions - Serverless для фронтенд-розробників:
vercel.com/docs/functions
- AWS Pricing Calculator - розрахуйте вартість своєї архітектури: **calculator.aws**

ПІДСУМОК ЛЕКЦІЇ

Поняття	Ключова думка	Що зробити прямо зараз
Архітектура хмарного додатку	Фундаментальне рішення, яке визначає всі можливості системи. Неправильна архітектура = катастрофічні наслідки для бізнесу.	Перед початком будь-якого проекту задайте питання: «Яке навантаження очікується? Як буде масштабуватись система?»
Serverless (без-серверна архітектура)	Код виконується тільки при запиті. 0 запитів = \$0 вартості. 0 управління серверами.	Напишіть Telegram-бот на AWS Lambda - безкоштовно для першого мільйона викликів
FaaS / AWS Lambda	Пишете одну функцію, AWS запускає її на подію. Оплата у мілісекундах. 1M запитів/міс - безкоштовно.	Зареєструйтесь та задеployте Hello World на Lambda за 15 хвилин
Cloud-Native (хмарно-орієнтований)	Додаток проектується для хмари: мікросервіси + Docker + автомасштабування + CI/CD. Netflix - еталон.	При розробці наступного проекту питайте: «Як це масштабуватиметься? Чи Stateless?»
Мікросервіси (Microservices)	Великий додаток = багато незалежних маленьких сервісів. Кожен масштабується і деплоїться окремо.	Поки що пишеть монолітні додатки (це правильно для MVP). Але розумійте, коли переходити.
12-Factor App (12-факторний)	12 принципів якісного хмарного додатку. Мінімум: Config у env vars, Stateless-процеси, Dev/Prod Parity.	Перевірте свій код прямо зараз - чи є у вас захардкожені (вписані в код) паролі?

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке архітектура хмарного додатку і яку роль вона відіграє у розробці системи?
2. Які основні питання вирішує хмарна архітектура при проектуванні системи?

3. Що таке Serverless-архітектура і в чому її головна ідея?
4. Чому Serverless не означає повну відсутність серверів?
5. У чому різниця між традиційним сервером і Serverless-підходом?
6. Які переваги Serverless-архітектури для стартапів і невеликих проєктів?
7. Що таке FaaS (Function as a Service) і як він пов'язаний із Serverless?
8. Що означає Stateless у контексті хмарних функцій?
9. Що таке Cold Start і чому він є проблемою для Serverless-додатків?
10. Які способи існують для зменшення впливу Cold Start?
11. Що таке Cloud-Native підхід і чим він відрізняється від Lift-and-Shift?
12. Що таке мікросервісна архітектура і які її основні переваги?
13. У чому різниця між монолітом і мікросервісами?
14. Що таке методологія 12-Factor App і для чого вона використовується?
15. Які принципи 12-Factor App є найважливішими для хмарних додатків і чому?

ЛЕКЦІЯ 4. GitHub та Netlify - сучасні інструменти розробника

Мета лекції

Сформувати у студентів чітке розуміння системи контролю версій Git та платформи GitHub, навчити публікувати сайти за допомогою Netlify. Пояснити, чим відрізняються Git і GitHub, як організувати командну роботу над кодом та автоматизувати розгортання проекту. Закласти практичну основу для самостійного розгортання (деплой) першого навчального проекту у хмарі.

План лекції

1. Git - система контролю версій: основні поняття та команди.
2. GitHub - від реєстрації до першого репозиторію: покрокова інструкція.
3. GitHub Student Developer Pack - безкоштовні переваги для студентів.
4. Тарифи GitHub - детальний огляд планів.
5. Netlify - хостинг для вашого сайту: деплой, CDN, Forms, Functions.
6. Тарифи Netlify - що безкоштовно, що платно.
7. CI/CD - автоматичний деплой: `git push` → сайт оновився.
8. Підсумок та рекомендована література.

Ключові терміни лекції:

Git - розподілена система контролю версій (Version Control System - VCS - система відстеження змін у коді).

GitHub - веб-сервіс для зберігання та спільної роботи над кодом на основі Git.

Repository (репозиторій, скорочено «репо») - папка-сховище коду з повною історією всіх змін.

Commit (коміт) - збережений «знімок» стану проекту з описом що було змінено.

Branch (гілка) - незалежна копія коду для розробки нової функції без впливу на основний код.

Pull Request (PR - запит на злиття) - пропозиція об'єднати зміни з однієї гілки в іншу.

CI/CD - Continuous Integration / Continuous Deployment (Безперервна Інтеграція / Безперервне Розгортання).

Netlify - PaaS-платформа (Platform as a Service - Платформа як послуга) для хостингу сайтів.

Deploy (деплой) - публікація (розгортання) готового сайту або додатку в Інтернеті.

CDN - Content Delivery Network (Мережа доставки контенту) - система серверів по всьому світу.

.gitignore - файл зі списком файлів і папок, які Git ігнорує і не відправляє на GitHub (паролі, кеш, бібліотеки).

ЩО TAKE GITHUB І ЧОМУ ЦЕ ВАЖЛИВО ЗНАТИ

Навіщо вивчати GitHub у дисципліні «Розгортання інформаційних систем»?

Перш ніж переходити до технічних деталей, дамо відповідь на головне питання: навіщо студенту першого курсу спеціальності 122 «Комп'ютерні науки» знати GitHub у межах саме цієї дисципліни?

«Розгортання інформаційних систем» - це дисципліна про те, як зробити так, щоб ваш код запрацював у реальному світі: не лише у вас на комп'ютері, а у хмарі, на сервері, у відкритому доступі в Інтернеті. Це шлях від ідеї до живого продукту. І цей шлях неможливий без двох ключових інструментів: системи контролю версій (Git/GitHub) та платформи розгортання (Netlify та аналогів).

GitHub - це не просто «де зберігати код». Це фундамент усього сучасного процесу розробки та розгортання. Без розуміння GitHub неможливо зрозуміти CI/CD (автоматичне розгортання), командну розробку, резервне копіювання коду, відстеження версій та безліч інших тем, які ви будете вивчати протягом усього курсу.

Конкретні зв'язки між GitHub і дисципліною:

- Без GitHub неможливо налаштувати автоматичне розгортання (CI/CD) - систему, де код сам публікується після кожного збереження.
- Без GitHub неможлива командна розробка - кілька людей не зможуть паралельно працювати над одним проектом.
- Без GitHub немає безпечного резервного копіювання - якщо зламається ноутбук, ваш код зникне назавжди.
- Без GitHub неможливо підключитись до Netlify, Vercel, Render та інших платформ розгортання.
- Без GitHub роботодавець не зможе побачити ваше портфоліо та оцінити ваш рівень.

Що таке GitHub: пояснення «з нуля»

Уявіть таку ситуацію: ви місяць пишете дипломний проект. Код займає кілька тисяч рядків, файлів десятки. І одного дня, додаючи нову функцію, ви випадково видаляєте щось важливе і не помічаєте цього. Зберігаєте. Закриваєте ноутбук. А наступного дня розумієте: тиждень роботи безповоротно знищено. Звична папка на робочому столі не зберегла попередню версію. Це трапляється з кожним розробником хоча б раз - і це дуже боляче.

Або інша ситуація: ви з трьома одногрупниками робите спільний проект. Домовились ділити файли через Telegram. Ігор зробив зміни у файлі index.html і поклав нову версію. Аліна, не знаючи про це, теж відредагувала той самий файл і замінила версію Ігора. Годину роботи Ігора - знищено. Починається конфлікт, ніхто не пам'ятає «яка версія правильна». Проект гальмує.

Git і GitHub вирішують обидві ці проблеми - і не лише їх.

Git - це програма, яку ви встановлюєте на свій комп'ютер. Вона запам'ятовує кожну зміну у вашому коді, як «машина часу». Ви можете в будь-який момент повернутись до стану проекту тижневої або місячної давності.

GitHub (github.com) - це веб-сервіс у хмарі, де ви зберігаєте ваші Git-репозиторії. Це як Google Drive, але спеціально для програмістів: з відстеженням змін, коментарями до коду, командними інструментами та автоматизацією розгортання.

Аналогія для розуміння: Git - це Microsoft Word з функцією «Відстеження змін» (показує що саме було змінено, ким і коли). GitHub - це Google Drive (місце в хмарі, де ваш код зберігається, синхронізується і доступний з будь-якого пристрою). Різниця: Git/GitHub зроблені спеціально для коду і набагато потужніші за звичайні хмарні сховища.

Що відбувається, якщо НЕ знати та не використовувати GitHub?

Це критично важливе питання. Незнання GitHub - це не просто «незручність». Це реальні наслідки, з якими стикаються тисячі початківців-розробників щороку.

НАСЛІДОК 1: Втрата коду - безповоротно і назавжди.

Без Git/GitHub ваш код живе лише в одному місці - на вашому комп'ютері. Жорсткий диск виходить з ладу в середньому раз на 3-5 років. Ноутбуки крадуть. Їх заливають кавою. Без резервної копії в GitHub - проект зникає. Студент, який не використовує GitHub, ризикує втратити дипломний проект за день до захисту. На форумі Reddit є тисячі постів з заголовком «I lost my entire project» - 90% цих людей не використовували Git.

НАСЛІДОК 2: «Пекло злиття» - нескінченні конфлікти в команді.

Без системи контролю версій командна розробка перетворюється на хаос. Четверо студентів одночасно редагують спільний проект через Telegram. Хтось надіслав «нову версію» - але в ній немає змін двох інших людей. Хтось «перезаписав» файл, не помітивши. В результаті - ніхто не знає, яка версія «актуальна», чиї зміни «загублені», і скільки часу витрачено даремно.

НАСЛІДОК 3: Неможливість відкату - «зламав і не можу виправити».

Ви додаєте нову функцію - і щось «зламалося». Раніше все працювало, а тепер - ні. Без Git ви навіть не знаєте, що саме ви змінили. Без Git ви не можете «відмотати» назад до стану

«коли все ще працювало». Єдиний варіант - вручну шукати помилку серед сотень рядків. З Git - одна команда, і ви повернулись до робочої версії за секунди.

НАСЛІДОК 4: Безпека - витік паролів і секретів.

Ті, хто не розуміє Git/GitHub, часто не знають про файл .gitignore і про те, що паролі, API-ключі та інші секрети НЕ ПОВИННІ потрапляти на GitHub. Щороку тисячі розробників випадково публікують секретні ключі у відкритих репозиторіях. Автоматизовані боти сканують GitHub і знаходять такі ключі за хвилини. Наслідки: зламаний акаунт, фінансові втрати, компрометація даних користувачів.

Реальний інцидент (2022 рік): Студент-розробник випадково закомітив файл з ключами доступу до Amazon AWS і відправив на публічний GitHub. За 3 хвилини автоматизований бот знайшов ключі й запустив 100 потужних серверів для майнінгу криптовалюти. Рахунок від Amazon за одну ніч: \$47,000 (сорок сім тисяч доларів). Правильне використання .gitignore рятує від таких катастроф.

НАСЛІДОК 5: Відсутність портфоліо - невидимість на ринку праці.

GitHub - це ваше публічне резюме як розробника. Рекрутери та технічні інтерв'юери ЗАВЖДИ просять посилання на GitHub-профіль. Студент без GitHub-профілю (або з порожнім профілем) виглядає так, ніби він нічого не робив за весь час навчання. Навіть якщо він написав десятки проектів - але не завантажив їх на GitHub - роботодавець цього не побачить.

НАСЛІДОК 6: Неможливість використовувати сучасні платформи розгортання.

Netlify, Vercel, Render, Railway - всі сучасні платформи для розгортання сайтів і додатків інтегруються з GitHub. Без GitHub ви не зможете використовувати жодну з них нормально. Ви будете вручну завантажувати файли через FTP, як це робили у 2005 році. Це повільно, ненадійно і не масштабується.

Порівняння: з GitHub та без GitHub

Ситуація	Без GitHub	З GitHub
Дипломний проект	Ризик втратити все при збої ноутбука	Автоматичний backup у хмарі, 100% безпека
Груповий проект	Конфлікти, «хто що зробив» невідомо	Паралельна робота без конфліктів, чітка історія
Пошук роботи / стажування	Немає чого показати рекрутеру	Живий профіль з проектами - ваше портфоліо
Публікація сайту	Вручну через FTP, повільно і незручно	git push → сайт оновлений за 60 секунд
Виправлення помилки	Невідомо що змінилось, пошук навамання	git log показує точно що і коли змінилось

GIT - СИСТЕМА КОНТРОЛЮ ВЕРСІЙ

Що таке Git і навіщо він потрібен

Уявіть, що ви місяць пишете дипломний проект - сайт на Python. Ви додаєте нову функцію реєстрації, і раптом весь проект перестає запускатись. Як повернутись до стану «коли ще все працювало»? Якщо ви просто зберігали файли у звичайній папці - ніяк. Тижні роботи можуть бути втрачені.

Git (Global Information Tracker - хоча сам автор каже, що це просто вигадане слово) - це програма, яка вирішує цю проблему. Вона запам'ятовує КОЖНУ зміну у вашому коді і дозволяє повернутись до будь-якого попереднього стану. Git був створений у 2005 році Лінусом Торвальдсом - тим самим, хто написав ядро операційної системи Linux. Git використовують 93% розробників у світі (за опитуванням Stack Overflow 2023).

Основні поняття Git - з простими поясненнями

Термін	Переклад	Простими словами	Приклад
Repository	Репозиторій (репо)	Папка вашого проекту, яку «спостерігає» Git. Всередині є прихована папка .git з усією історією змін.	Папка my-website/ зі всіма файлами сайту
Commit	Коміт - фіксація/збереження	«Знімок» всього проекту у певний момент. Як кнопка «Зберегти», але зі збереженням попередніх версій.	«Додав сторінку контактів» або «Виправив баг у формі входу»
Branch	Гілка	Копія коду для розробки нової функції. Ви працюєте на окремій гілці, не чіпаючи основний код. Коли готово - зливаєте назад.	Гілка feature/registration для розробки реєстрації
Merge	Мердж - злиття гілок	Об'єднання змін з однієї гілки в іншу. Git намагається зробити це автоматично.	Злиття гілки feature/registration у main
Clone	Клон - копіювання репо	Завантаження повної копії репозиторію (з усією історією) з GitHub на ваш комп'ютер.	git clone https://github.com/user/project.git
Push	Пуш - відправка коду	Відправити ваші коміти з локального комп'ютера на GitHub (у хмару).	git push - після цього ваш код видно на github.com

Pull	Пул - отримання коду	Завантажити нові зміни з GitHub на ваш комп'ютер.	git pull - «синхронізація» з GitHub
.gitignore	Гітігнор	Файл із переліком файлів/папок, які Git повинен ІГНОРУВАТИ. Паролі, кеш, бібліотеки - туди.	node_modules/, .env, *.pyc

Базові команди Git - мінімум, який треба знати

Вам не потрібно знати всі 200+ команд Git. Нижче - ті 10 команд, якими ви будете користуватись щодня. Все інше вивчите поступово в процесі роботи.

```
# ВСТАНОВЛЕННЯ GIT (виконується один раз)
# Windows: завантажте інсталятор на https://git-scm.com/download/win
# Mac: у терміналі введіть: brew install git
# Ubuntu/Debian Linux: sudo apt install git

# ПЕРШЕ НАЛАШТУВАННЯ (виконується один раз після встановлення)
git config --global user.name "Іван Петренко" # ваше ім'я
git config --global user.email "ivan@example.com" # ваш email

# ПОЧАТОК РОБОТИ З ПРОЕКТОМ
git init # зробити поточну папку репозиторієм
git clone https://github.com/user/repo.git # завантажити репо з GitHub

# ЩОДЕННИЙ ЦИКЛ РОБОТИ (запам'ятайте цю послідовність!)
git status # перевірити: які файли змінені?
git add . # позначити ВСІ змінені файли
git add index.html # або тільки один конкретний файл
git commit -m "Короткий опис що зробили" # зберегти знімок з описом
git push origin main # відправити на GitHub

# ОТРИМАННЯ ЗМІН ВІД КОМАНДИ
git pull origin main # завантажити нові зміни з GitHub

# ПЕРЕГЛЯД ІСТОРІЇ
git log --oneline # скорочена історія комітів
git log # повна детальна історія
```

Правило хорошого повідомлення до коміту. Повідомлення відповідає на питання: «Що тепер вміє/робить цей код?»

• ПОГАНО: `git commit -m "зміни"` або `git commit -m "fix"` або `git commit -m "аааа"` нарешті працює"

- ДОБРЕ: `git commit -m "feat: додав форму реєстрації користувача"`
- ДОБРЕ: `git commit -m "fix: виправив помилку при введенні пустого email"`
- ДОБРЕ: `git commit -m "style: змінив колір кнопки на синій"`

Префікси за угодою Conventional Commits: `feat`: - нова функція; `fix`: - виправлення помилки; `docs`: - документація; `style`: - форматування без змін логіки; `refactor`: - покращення коду без зміни функціональності.

Файл `.gitignore` - що НЕ повинно потрапити на GitHub

Файл `.gitignore` (назва починається з крапки - це «прихований» файл у Linux/Mac) - один із найважливіших файлів у будь-якому проєкті. Він містить список файлів і папок, які Git повинен повністю ігнорувати - не відстежувати і не відправляти на GitHub.

Чому це критично важливо:

- **Безпека:** паролі, API-ключі, токени не повинні потрапити у публічний репозиторій.
- **Розмір:** папка `node_modules` у JavaScript-проєкті може важити 300+ МБ - нема сенсу зберігати її у Git, бо `npm install` відновить її за секунди.
- **Чистота:** тимчасові файли редакторів, кеш системи - не потрібні іншим розробникам.

```
# Файл .gitignore - кладемо у кореневу папку проєкту

# СЕКРЕТИ ТА ПАРОЛІ - НАЙВАЖЛИВІШЕ!
.env                # файл зі змінними середовища (паролі, API_KEY тощо)
.env.local           # локальні змінні середовища
.env.production      # змінні для продуктивного середовища
# ЗАЛЕЖНОСТІ (БІБЛІОТЕКИ) - ВАЖАТЬ СОТНІ МБ
node_modules/        # Node.js пакети (відновлюються: npm install)
venv/                # Python virtual environment
__pycache__/         # кешовані Python-файли
*.pyc                # скомпільовані Python-файли
# ЗБІРКИ (BUILD - РЕЗУЛЬТАТ КОМПІЛЯЦІЇ КОДУ)
dist/                # папка з готовою збіркою для React/Vue тощо
build/
.next/               # збірка Next.js

# ФАЙЛИ РЕДАКТОРА ТА ОС
.vscode/settings.json # особисті налаштування VS Code
.DS_Store            # системний файл macOS
Thumbs.db            # системний файл Windows
*.log                # log-файли (журнал подій програм)
```

Порада: при створенні нового репозиторію на GitHub є галочка «Add .gitignore» - оберіть мову вашого проекту і GitHub автоматично створить правильний .gitignore. Також є сайт [gitignore.io](https://github.com/gitignore.io) - вводите назву мови і отримуєте готовий файл.

GITHUB - ВІД РЕЄСТРАЦІЇ ДО ПЕРШОГО ДЕПЛОЮ

Що таке GitHub і чим він відрізняється від Git

Git і GitHub - це НЕ одне й те саме, і ця плутанина дуже поширена серед початківців. Git - це програма, яку ви встановлюєте на свій комп'ютер. GitHub (github.com) - це веб-сайт у хмарі, де ви зберігаєте свої Git-репозиторії та спільно працюєте з іншими.

Аналогія: Git - це Microsoft Word (програма для редагування документів), а GitHub - це Google Drive (хмарне сховище, де ви ці документи зберігаєте і ділитесь ними). Ви можете використовувати Git без GitHub (тільки локально), але без Git ви не зможете користуватись GitHub у повній мірі.

GitHub заснований у 2008 році, у 2018-му куплений Microsoft за \$7.5 млрд. Зараз на GitHub зберігається понад 420 мільйонів репозиторіїв. На GitHub «живе» практично весь відкритий код (open source): Linux, Python, React, VS Code, Android - все там.

Чому GitHub важливий особисто для вас як студента:

- **Портфоліо:** GitHub - це ваше публічне резюме коду. Роботодавці дивляться на ваш GitHub-профіль при відборі кандидатів. Студент з 10 проектами на GitHub виглядає значно сильніше.

- **Безпека коду:** ваш дипломний проект не зникне якщо зламається ноутбук. GitHub - це автоматичний backup у хмарі.

- **Спільна робота:** без GitHub зробити груповий проект - це пекло. З GitHub кілька людей паралельно вносять зміни без конфліктів.

- **Безкоштовні інструменти:** через GitHub Student Pack ви отримуєте інструменти вартістю \$200+ на місяць абсолютно безкоштовно.

Інтерфейс GitHub - що де знаходиться

Елемент інтерфейсу	Де знаходиться	Що це таке і навіщо
Profile (профіль)	Клік на ваш аватар → Your profile	Ваша публічна сторінка з усіма репозиторіями, contribution graph (графік активності) і bio (короткий опис про себе).

Repositories (репозиторії)	Вкладка Repositories у профілі	Список всіх ваших репозиторіїв - публічних і приватних.
Stars (зірки)	Кнопка Star на будь-якому репозиторії	Як «лайк» у соцмережах - позначає репозиторії, що сподобались.
Fork (форк)	Кнопка Fork у правому верхньому куті репозиторію	Створення власної копії чужого репозиторію у вашому акаунті. Ви можете змінювати форк і пропонувати зміни назад оригіналу.
Issues (задачі/баги)	Вкладка Issues у репозиторії	Система відстеження задач і помилок. Схоже на Jira або Trello, але вбудоване прямо у репозиторій.
Pull Requests (PR)	Вкладка Pull requests у репозиторії	Пропозиція об'єднати зміни з однієї гілки в іншу. Основний механізм перевірки коду в команді.
Actions (дії)	Вкладка Actions у репозиторії	GitHub Actions - CI/CD. Автоматичне виконання задач при events (подіях): push, PR тощо.
Settings (налаштування)	Вкладка Settings у репозиторії	Налаштування видимості (public/private), GitHub Pages, collaborators (співавтори).

Покрокова інструкція: від реєстрації до першого репозиторію

Крок 1: Реєстрація на GitHub

1. Відкрийте github.com у браузері.
2. Натисніть «Sign up» (zareєstruvatis'ya) у правому верхньому куті.
3. Введіть email адресу - ВАЖЛИВО: використовуйте університетський email (.edu.ua) для отримання Student Pack.
4. Придумайте пароль (мінімум 15 символів або 8 символів + цифри).
5. Введіть username (ім'я користувача) - це частина вашого URL: github.com/YOUR_USERNAME. Оберіть щось професійне, наприклад ivan-petrenko або ivanpetrenko. Це будуть бачити роботодавці!
6. Підтвердіть email - перейдіть до поштової скриньки і клікніть на посилання підтвердження.
7. Готово! Ваш профіль на github.com/YOUR_USERNAME вже доступний.

Крок 2: Створення першого репозиторію

1. На головній сторінці GitHub натисніть зелену кнопку «New» або кнопку «+» → «New repository».

2. Заповніть форму: Repository name (назва): my-first-website (використовуйте дефіси замість пробілів!); Description (опис): «Мій перший сайт на HTML/CSS»; Visibility (видимість): Public (публічний) - видно всім, або Private (приватний) - тільки вам.
3. Поставте галочку «Add a README file» - README.md буде головна сторінка репо з описом.
4. Виберіть «Add .gitignore» → оберіть шаблон для вашої мови (наприклад Python або Node).
5. License (ліцензія): для навчальних проектів зазвичай MIT License.
6. Натисніть «Create repository» (створити репозиторій). Готово!
7. Скопіюйте HTTPS URL (виглядає як `https://github.com/YOUR_NAME/my-first-website.git`).

Крок 3: Прив'язка локального проекту до GitHub

Відкрийте термінал у папці вашого проекту і виконайте по черзі:

```
git init                                # ініціалізуємо Git у папці проекту
git add .                               # додаємо всі файли до першого коміту
git commit -m "init: перший коміт"      # зберігаємо перший знімок

# Прив'язуємо локальну папку до GitHub-репозиторію:
# origin - це стандартна назва для remote (віддаленого) репозиторію
git remote add origin https://github.com/YOUR_NAME/my-first-website.git

# Відправляємо код на GitHub:
# -u означає 'set upstream' - встановити відстеження (тільки першого разу)
git push -u origin main

# Після цього достатньо писати просто: git push
# Відкрийте github.com/YOUR_NAME/my-first-website - ваш код вже там!
```

Гілки (Branches) - паралельна розробка

Уявіть, що ви написали сайт і він добре працює. Тепер вам потрібно додати сторінку «Контакти» - це займе кілька днів і в процесі роботи код буде «зламаний». Як при цьому зберегти робочу версію сайту? Відповідь - гілки (branches).

Гілка main (або master у старіших проектах) - це завжди стабільна, робоча версія коду. Для кожної нової функції або виправлення ви створюєте окрему гілку, там пишете код (навіть якщо він тимчасово зламаний), і лише після того як все перевірили - зливаєте у main.

```

git branch                                # переглянути список всіх гілок (* =
поточна)

git branch feature/contacts-page          # створити нову гілку
git checkout feature/contacts-page        # переключитись на гілку
git checkout -b feature/contacts-page     # створити і переключитись
(найзручніше)

# Тепер ви на гілці feature/contacts-page
# Робіть зміни, коміти - вони не торкаються гілки main
git add .
git commit -m "feat: додав сторінку контактів"
git push origin feature/contacts-page     # відправити гілку на GitHub

# Злиття гілки у main
git checkout main                         # повернутись на main
git merge feature/contacts-page           # злити зміни з feature-гілки
git push origin main                     # відправити оновлений main на
GitHub

# Видалення гілки після злиття
git branch -d feature/contacts-page       # видалити локальну гілку

```

Pull Requests (PR) - запити на злиття коду

Pull Request (PR - запит на злиття) - це спосіб запропонувати свої зміни і попросити когось їх перевірити перед злиттям у main. У реальних компаніях НІХТО не пушить код напряму у main. Всі зміни проходять через PR і code review (перевірку коду).

Крок	Дія	Де виконується	Що відбувається
1	Розробник пушить гілку на GitHub	Термінал: <code>git push origin feature/login</code>	Гілка з'являється на GitHub.
2	Відкриває Pull Request	GitHub → кнопка «Compare & pull request»	Описує що зробив, чому, як тестував. Можна додати скріншоти.
3	Колеги або викладач роблять Code Review	GitHub → вкладка Files changed	Залишають коментарі до конкретних рядків коду.
4	Розробник вносить правки	Локально + <code>git push</code>	Нові коміти автоматично з'являються у відкритому PR.
5	PR затверджується (Approve)	GitHub → кнопка «Approve»	Колега підтверджує: «код перевірений, виглядає добре».
6	Merge (злиття) у main	GitHub → кнопка «Merge pull request»	Зміни потрапляють у main, гілку можна видалити.

GitHub Pages - безкоштовний хостинг прямо з репозиторію

GitHub Pages - це безкоштовний хостинг статичних сайтів, вбудований прямо у GitHub. Ваш сайт автоматично публікується з репозиторію за адресою USERNAME.github.io/REPO_NAME. Жодних налаштувань серверів, жодних кредитних карток - просто увімкнули функцію і сайт онлайн.

GitHub Pages підтримує: чистий HTML, CSS, JavaScript - будь-який статичний сайт; Jekyll (генератор статичних сайтів від GitHub); React, Vue, Angular - якщо попередньо зібрати через npm run build; сайт-портфоліо розробника. НЕ підтримує: серверний код (Python Flask, Node.js Express, PHP) - тільки статика; бази даних; файли розміром понад 100 МБ.

Як увімкнути GitHub Pages:

1. Відкрийте ваш репозиторій на GitHub.
2. Перейдіть: Settings (налаштування) → Pages (сторінки) - у лівому меню.
3. У розділі «Source» (джерело) оберіть: «Deploy from a branch» (розгорнути з гілки).
4. Оберіть гілку: main, папку: / (root - корінь репозиторію).
5. Натисніть Save (зберегти).
6. Зачекайте 1-2 хвилини і перейдіть за адресою: https://YOUR_USERNAME.github.io/REPO_NAME
7. Сайт онлайн! При кожному git push у main GitHub автоматично оновлює сайт.

GitHub Actions - автоматизація задач

GitHub Actions (дії GitHub) - це вбудований інструмент для CI/CD. Він дозволяє автоматично виконувати будь-які задачі у відповідь на events (події) у репозиторії: push, відкриття PR, розклад (schedule) тощо. Безкоштовно - 2000 хвилин на місяць для публічних репозиторіїв.

```
# Файл: .github/workflows/deploy.yml
# Ця автоматизація запускається при кожному push у гілку main
name: Deploy to GitHub Pages
on:
  push:
    branches:
      - main
jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
```

```
- name: Checkout code
  uses: actions/checkout@v4
- name: Setup Node.js
  uses: actions/setup-node@v4
  with:
    node-version: '20'
- name: Install and Build
  run: |
    npm install
    npm run build
- name: Deploy
  uses: peaceiris/actions-gh-pages@v4
  with:
    github_token: ${ secrets.GITHUB_TOKEN }
    publish_dir: ./dist
```

GITHUB STUDENT DEVELOPER PACK - БЕЗКОШТОВНІ ПЕРЕВАГИ ДЛЯ СТУДЕНТІВ

Що таке GitHub Student Developer Pack

GitHub Student Developer Pack - це програма, яка надає студентам безкоштовний доступ до десятків платних інструментів і сервісів, якими користуються професійні розробники. Загальна вартість усіх інструментів у пакеті перевищує \$200 на місяць або \$2,400+ на рік. Все це доступно вам безкоштовно, поки ви є студентом.

Найцінніші складові пакету:

- **GitHub Pro** - необмежені приватні репозиторії, розширені Insights, захищені гілки. Звичайна ціна: \$4/місяць.

- **GitHub Copilot** (AI-помічник з написання коду, вбудований у VS Code та JetBrains) - автодоповнення цілих функцій на основі коментаря чи назви. Безкоштовно для студентів! Звичайна ціна: \$10/місяць.

- **Microsoft Azure** (хмарна платформа від Microsoft) - \$100 кредитів на хмарні сервіси.

- **JetBrains** (IDE): PyCharm, WebStorm, IntelliJ IDEA, DataGrip - всі продукти безкоштовно. Звичайна ціна: \$249/рік.

- **Namecheap** - безкоштовний домен .me на 1 рік (наприклад: ivanpetrenko.me). Звичайна ціна: ~\$10/рік.

- **DigitalOcean** - \$200 кредитів на хмарні сервери на 1 рік.

- **DataCamp** - курси з Data Science та Machine Learning безкоштовно.

Покрокова інструкція: як підключити GitHub Student Pack

Що потрібно для підключення: акаунт на GitHub; університетський email (.edu.ua або подібний) - НАЙПРОСТІШИЙ спосіб: якщо у вас є email виду student@nau.edu.ua або ivan@knu.ua - GitHub автоматично підтверджує статус студента без додаткових документів; якщо немає університетського email - студентський квиток (фото або скан), довідка з університету про навчання.

КРОК 1 - Додайте університетський email до акаунту GitHub:

- Натисніть на ваш аватар → Settings → Emails → «Add email address» → введіть університетський email і підтвердіть через лист.

КРОК 2 - Перейдіть на сторінку заявки:

- Відкрийте: education.github.com/discount_requests/application

КРОК 3 - Заповніть заявку:

- «What best describes your academic status?» → оберіть «Student».
- «What is the name of your school?» → введіть назву університету.
- «How do you plan to use GitHub?» → опишіть ваші плани (2-3 речення).

КРОК 4 - Оберіть спосіб верифікації:

- ВАРІАНТ А (найшвидший): якщо GitHub розпізнав ваш університетський email - натисніть «Verify school email».
- ВАРІАНТ Б: верифікація через документ - завантажте фото або скан студентського квитка (JPG, PNG або PDF).

КРОК 5 - Очікуйте результату:

- При верифікації через email: підтвердження миттєво або протягом кількох хвилин.
- При верифікації через документ: від кількох годин до 3-5 робочих днів.
- Прийде email з результатом на вашу адресу.

ТАРИФИ GITHUB - ДЕТАЛЬНИЙ ОГЛЯД

Огляд тарифних планів GitHub

GitHub має чотири основні тарифні плани. Для студентів і більшості особистих проєктів безкоштовного плану (Free) цілком достатньо, а через Student Pack ви отримуєте Pro безкоштовно.

Функція	Free (безкоштовний)	Pro (\$4/міс або безкоштовно зі Student Pack)	Team (\$4/людину/міс)	Enterprise (\$21/людину/міс)
Публічні репозиторії	Необмежено	Необмежено	Необмежено	Необмежено

Приватні репозиторії	Необмежено	Необмежено	Необмежено	Необмежено
GitHub Actions хв/міс	2,000 хв/міс	3,000 хв/міс	3,000 хв/міс	50,000 хв/міс
Protected branches (захищені гілки)	Немає	Є	Є	Є
GitHub Pages	Публічні репо	Публічні + приватні репо	Є	Є
GitHub Copilot	\$10/міс окремо	Безкоштовно зі Student Pack!	\$10/міс	Включено
Кому підходить	Студенти, особисті проекти	Студенти зі Student Pack, фрілансери	Невеликі команди та стартапи	Великі корпорації

Для навчання на 1-му курсі та виконання домашніх завдань - FREE плану повністю достатньо. FREE дає вам: необмежену кількість публічних і приватних репозиторіїв; 2,000 хвилин GitHub Actions на місяць (більш ніж достатньо); GitHub Pages (безкоштовний хостинг статичних сайтів); всі основні функції Git. Але GitHub Student Pack (PRO безкоштовно) варто підключити відразу, бо GitHub Copilot значно пришвидшує написання коду, а JetBrains IDE - найкращі редактори для Python і JS.

NETLIFY - ХОСТИНГ ДЛЯ ВАШОГО САЙТУ

Що таке Netlify і як він працює?

Netlify (netlify.com) - це PaaS-платформа (Platform as a Service - Платформа як послуга), яка спеціалізується на хостингу статичних сайтів і JAMstack-додатків. Заснована у 2015 році, зараз обслуговує понад 5 мільйонів сайтів і 3 мільйони розробників.

Ключова ідея Netlify: ви підключаєте ваш GitHub-репозиторій, Netlify автоматично публікує ваш сайт і ОНОВЛЮЄ його при кожному git push. Жодних серверів, жодного FTP, жодних ручних оновлень. Щойно ви відправили новий код - через 1-2 хвилини він вже на сайті.

Що таке CDN і чому завдяки ньому Netlify такий швидкий. CDN (Content Delivery Network - Мережа доставки контенту) - це мережа серверів, розташованих у десятках країн по всьому світу. Коли ви публікуєте сайт на Netlify, він автоматично копіюється на 100+ серверів: в Україні, Польщі, Німеччині, США, Японії, Бразилії. Коли студент з Харкова відкриває ваш сайт - без CDN запит іде до сервера у США (500-1000 мс затримки), з CDN (Netlify) - запит іде до найближчого сервера (наприклад, Варшава, 50-80 мс). Результат: сайт на Netlify

завантажується у 5-10 разів швидше ніж на звичайному хостингу. І все це безкоштовно включено у Netlify Free план!

Повний огляд можливостей Netlify

Можливість	Що це?	Навіщо студенту?	Безкоштовно?
Static Hosting	Розміщення HTML/CSS/JS файлів на глобальній CDN-мережі	Опублікувати сайт-портфоліо, дипломний проект, лендинг	Так, без обмежень за кількістю сайтів
Continuous Deployment	Автоматичне оновлення сайту при кожному git push до GitHub	git push → сайт оновився за 60 секунд. Не потрібно вручну завантажувати файли.	Так
Custom Domain	Підключення вашого домену замість site.netlify.app	Виглядає професійно в портфоліо. Домен .me безкоштовно через Student Pack.	Так (SSL/TLS включено)
HTTPS / SSL/TLS	Автоматичний безкоштовний сертифікат безпеки	Без HTTPS браузері показують «небезпечний сайт». Netlify робить це автоматично.	Так, автоматично
Deploy Previews	Окремий URL для кожного Pull Request	Показати викладачу або замовнику результат ДО оновлення основного сайту.	Так
Netlify Forms	Обробка HTML-форм без написання серверного коду	Додати форму «напишіть мені» на портфоліо без бекенду.	100 відправок/міс
Netlify Functions	Serverless functions - Node.js код при HTTP-запиті	Надсилати email, обробляти платежі без власного сервера.	125,000 запитів/міс
Analytics	Вбудована аналітика відвідувачів без сторонніх скриптів	Скільки людей зайшло на ваш сайт, звідки, які сторінки переглядали.	Базова безкоштовно

Покрокова інструкція: перший деплой на Netlify

Спосіб 1 - Підключення до GitHub (рекомендований)

1. Перейдіть на netlify.com і натисніть «Sign up» → «Sign up with GitHub».
2. Дайте Netlify дозвіл на доступ до ваших репозиторіїв.
3. Натисніть «Add new site» → «Import an existing project» → «Deploy with GitHub».
4. Зі списку ваших репозиторіїв оберіть потрібний (наприклад my-first-website).

5. Налаштуйте «Build settings»: Branch to deploy - main; Build command - залиште ПОРОЖНІМ для чистого HTML (для React: npm run build); Publish directory - / (корінь) для чистого HTML (для React/Vite: dist).
6. Натисніть «Deploy site» (розгорнути сайт).
7. Зачекайте 30-90 секунд - сайт опублікований! Netlify дав автоматичну адресу виду: random-words-12345.netlify.app
8. Тепер кожен git push у main → Netlify автоматично оновлює сайт!

Спосіб 2 - Drag & Drop: для швидкого тесту

Це найшвидший спосіб - менше 30 секунд від ідеї до живого сайту. Але він НЕ має автоматичного оновлення при змінах коду.

- Відкрийте netlify.com у браузері (можна без реєстрації!).
- На головній сторінці є область із написом «Drag and drop your site output folder here».
- Перетягніть ПАПКУ (не окремий файл!) у браузерне вікно Netlify.
- Netlify миттєво публікує і дає URL - сайт онлайн!

Netlify Forms - форма без серверу

Одна з найкорисніших функцій Netlify для студентів - обробка HTML-форм без будь-якого серверного коду. Зазвичай, щоб форма «Напишіть мені» надсилала дані, потрібен бекенд на Python або Node.js. З Netlify Forms достатньо одного додаткового атрибута у HTML!

```
<!-- Звичайна HTML-форма - НІЧОГО НЕ РОБИТЬ без серверу -->
<form method="POST" action="/submit">
  <input type="text" name="name" required>
  <input type="email" name="email" required>
  <textarea name="message" required></textarea>
  <button type="submit">Надіслати</button>
</form>

<!-- Та САМА форма але з Netlify Forms - достатньо додати data-netlify="true" -->
<form name="contact" method="POST" data-netlify="true">
  <input type="hidden" name="form-name" value="contact">
  <input type="text" name="name" required>
  <input type="email" name="email" required>
  <textarea name="message" required></textarea>
  <button type="submit">Надіслати</button>
</form>
```

```
<!-- Netlify автоматично:
  - перехоплює відправку форми
  - зберігає дані у вашому Netlify Dashboard
  - надсилає вам email-повідомлення при кожній відправці -->
```

ТАРИФИ NETLIFY - ЩО БЕЗКОШТОВНО, ЩО ПЛАТНО

Огляд тарифних планів Netlify

Функція	Starter Безкоштовний	Pro - \$19/міс	Enterprise - за домовленістю
Кількість сайтів	Необмежено	Необмежено	Необмежено
Bandwidth (обсяг трафіку/міс)	100 ГБ - більш ніж достатньо для навчального проекту	1 ТБ (терабайт)	Необмежено
Build minutes (хвилини збірки/міс)	300 хв/міс (~100 деплоїв/день)	25,000 хв/міс	Необмежено
Netlify Functions	125,000 запитів/міс	2 млн запитів/міс	Необмежено
Netlify Forms	100 відправок/міс	19,000 відправок/міс	Необмежено
Deploy Previews	Є для кожного PR	Є	Є
Custom domain + HTTPS	Безкоштовно!	Є	Є
Password protection	Немає	Є	Є
Кому підходить?	Студенти, особисті проекти, портфоліо	Малий бізнес, фрілансери, агенції	Великі компанії

Для 1-го курсу безкоштовного плану Netlify більш ніж достатньо. 300 хвилин збірки - це приблизно 100-150 деплоїв на місяць; 100 ГБ трафіку - при 100 відвідувачах/день більш ніж вистачить; всі основні функції доступні: HTTPS, CDN, Deploy Previews, Forms, Functions. Перший раз коли вам знадобиться платний план - це дуже не скоро.

CI/CD - АВТОМАТИЧНИЙ ДЕПЛОЙ: GIT PUSH → САЙТ ОНОВИВСЯ

Що таке CI/CD і навіщо це студенту?

CI/CD (Continuous Integration / Continuous Deployment - Безперервна Інтеграція / Безперервне Розгортання) - це практика автоматизації, при якій код після кожного git push

автоматично перевіряється (тести), збирається (build) і публікується (deploy). Замість ручних дій - все відбувається само.

Базовий CI/CD налаштовується автоматично при підключенні GitHub до Netlify - без будь-якого написання YAML-файлів. Щойно ви підключили GitHub-репозиторій до Netlify - CI/CD вже працює: git push → Netlify отримує нову версію → збирає → публікує.

Порівняння: без CI/CD і з CI/CD:

- **БЕЗ CI/CD - ручний деплой (15-30 хвилин кожного разу):** написали код → відкрили FTP-клієнт → вручну завантажили файли → перевірили → забули скопіювати один файл → сайт зламаний → повторюємо спочатку.

- **З CI/CD через Netlify + GitHub (0 хвилин вашого часу):** написали код → git push → через 60 секунд сайт оновлений - отримали email з підтвердженням.

Як GitHub Actions + Netlify працюють разом

Базовий CI/CD (тільки деплой) налаштовується автоматично при підключенні GitHub до Netlify. Але якщо ви хочете ТАКОЖ запускати автоматичні тести перед деплоєм - потрібен файл GitHub Actions.

```
# Файл: .github/workflows/ci.yml
name: CI - Перевірка коду перед деплоєм
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
jobs:
  test:
    name: Запуск тестів
    runs-on: ubuntu-latest
    steps:
      - name: "1. Завантажити код"
        uses: actions/checkout@v4

      - name: "2. Встановити Node.js"
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'

      - name: "3. Встановити залежності"
        run: npm ci
```

```

- name: "4. Перевірити код лінтером"
  run: npm run lint

- name: "5. Запустити тести"
  run: npm test

- name: "6. Зібрати проект"
  run: npm run build

```

Якщо будь-який крок завершився помилкою - GitHub зупиняє виконання

і Netlify НЕ отримує сигнал на деплой → сайт не оновиться із зламаним кодом!

Статуси деплоїв у Netlify - що означають кольори

Статус Netlify	Колір/іконка	Що відбувається	Що робити
Building (збирається)	Жовтий / spinner	Netlify виконує вашу build command. Наприклад: <code>npm run build</code> .	Зачекати 1-3 хвилини
Published (опубліковано)	Зелений (чек)	Деплой успішний! Сайт оновлено, нова версія доступна за вашим URL.	Перейти на сайт і перевірити
Failed (не вдалось)	Червоний (x)	Збірка завершилась помилкою. Зміни НЕ опубліковані, стара версія продовжує працювати.	Клікнути на деплой → переглянути Build log → знайти рядок з ERROR
Cancelled (скасовано)	Сірий	Деплой був вручну скасований або замінений новішим деплоєм.	Зазвичай нічого не треба робити
Skipped (пропущено)	Сірий	Netlify виявив що код не змінився від попереднього деплою і пропустив збірку.	Нічого - це оптимізація для економії build minutes

Як знайти помилку у Failed деплої на Netlify:

1. Відкрийте Netlify Dashboard → ваш сайт.
2. Вкладка «Deploys» (деплої) → клікніть на червоний деплой зі статусом Failed.
3. Ви побачите «Build log» (журнал збірки) - довгий список команд що виконувались.

4. Шукайте рядки що починаються на: ERROR, error, Failed, npm ERR!, failed to.
5. Поруч з помилкою зазвичай є пояснення що пішло не так.

Найчастіші причини Failed деплою: синтаксична помилка у коді (забули дужку, кому тощо); бібліотека у коді є, але не записана у package.json або requirements.txt; неправильно вказана «Publish directory» у налаштуваннях; build command повертає помилку (спробуйте запустити її локально спочатку).

ПІДСУМОК ЛЕКЦІЇ

Ця лекція охопила два фундаментальних інструменти сучасного розробника - GitHub для зберігання коду і спільної розробки, та Netlify для публікації сайтів.

#	Ключова думка	Практичний висновок
1	Git - машина часу для коду. Кожен commit = знімок проекту. Можна повернутись до будь-якого моменту.	Зробіть git commit після кожної логічно завершеної зміни. Commit часто - це добре!
2	GitHub - хмарне сховище + портфоліо. Роботодавці дивляться на ваш GitHub-профіль.	Відразу після реєстрації заповніть профіль: фото, біо, закріпіть кращі репозиторії.
3	GitHub Student Pack дає PRO + Copilot + JetBrains + домен + \$400 кредитів безкоштовно.	Підключіть ЗАРАЗ на education.github.com/pack . Займає 5-10 хвилин.
4	Netlify = git push → сайт оновлений. CDN по всьому світу. HTTPS. Форми. Все безкоштовно.	Для будь-якого сайту без серверного коду - Netlify є найкращим вибором.
5	.gitignore - обов'язковий файл. Паролі, API-ключі, node_modules - НІКОЛИ не в GitHub.	Перевірте ваш .gitignore перед першим push. Один витік пароля може коштувати дорого.
6	CI/CD - автоматизація деплою. git push → тести → деплой → сайт оновився.	З Netlify CD вже налаштований автоматично. GitHub Actions для CI потрібен тільки для тестів.
7	Free плани GitHub і Netlify достатні для всього 1-го курсу і дипломного проекту.	Не платіть за хостинг поки навчаєтесь. GitHub Free + Netlify Free = \$0 і повний функціонал.

КОНТРОЛЬНІ ПИТАННЯ

Блок 1. Git -система контролю версій

1. Що таке Git і яку проблему він вирішує? Наведіть конкретний приклад ситуації, де відсутність Git призводить до втрати роботи.

2. Поясніть різницю між поняттями *repository*, *commit* і *branch*. Наведіть приклад використання кожного.
3. Що таке гілка (branch) і навіщо вона потрібна? Опишіть сценарій, коли робота без гілок призведе до проблем.
4. Назвіть та поясніть не менше п'яти базових команд Git, якими розробник користується щодня.
5. Що таке Pull Request? Опишіть повний цикл роботи з PR від відкриття до злиття у main.
6. Що таке файл .gitignore і яка його роль? Які три категорії файлів обов'язково потрібно додавати до .gitignore?
7. Що таке «Conventional Commits» і які префікси використовуються для позначення типів змін? Чому важливо писати змістовні повідомлення до комітів?

Блок 2. GitHub -від реєстрації до першого репозиторію

8. Чим відрізняється Git від GitHub? Поясніть за допомогою аналогії з відомими програмами.
9. Перерахуйте не менше п'яти реальних наслідків, до яких призводить незнання або ігнорування GitHub у роботі розробника.
10. Що таке GitHub Pages? Які типи сайтів він підтримує, а які -ні?
11. Які кроки необхідно виконати, щоб прив'язати локальну папку проекту до репозиторію на GitHub?
12. Що таке GitHub Actions? Для чого він використовується та скільки хвилин роботи надається безкоштовно?

Блок 3. GitHub Student Developer Pack

13. Що таке GitHub Student Developer Pack і яку загальну вартість інструментів він надає безкоштовно?
14. Назвіть не менше чотирьох складових Student Pack та вкажіть їхню звичайну вартість.
15. Яка умова є найшвидшим способом підтвердити статус студента при реєстрації у Student Pack?

Блок 4. Тарифи GitHub

16. Порівняйте плани GitHub Free та GitHub Pro: чим вони відрізняються за функціями GitHub Actions та GitHub Pages?
17. Чи достатньо плану GitHub Free для виконання завдань першого курсу? Обґрунтуйте відповідь.

Блок 5. Netlify -хостинг для сайту

18. Що таке Netlify і яка його ключова ідея щодо автоматизації публікації сайтів?

19. Що таке CDN і чому завдяки йому сайти на Netlify завантажуються швидше, ніж на звичайному хостингу?

20. Опишіть покроково, як задеплоїти сайт на Netlify через підключення до GitHub.

21. Що таке Netlify Forms і яку перевагу вона дає студенту при створенні портфоліо?

22. Що таке Netlify Functions і для яких завдань вони використовуються?

Блок 6. Тарифи Netlify

23. Які обмеження має безкоштовний план Netlify (Starter) за трафіком, хвилинами збірки та формами?

24. Чи достатньо безкоштовного плану Netlify для навчального проекту першого курсу? Обґрунтуйте.

Блок 7. CI/CD -автоматичний деплой

25. Розшифруйте аббревіатуру CI/CD та поясніть, що означає кожна з частин.

26. Порівняйте ручний деплой (без CI/CD) і автоматичний (з Netlify + GitHub). У чому переваги автоматизації?

27. Що означають статуси деплою на Netlify: *Building*, *Published*, *Failed*, *Skipped*? Що слід робити при статусі *Failed*?

28. Назвіть чотири найпоширеніші причини невдалого (Failed) деплою на Netlify.

Питання підвищеної складності

29. Студент випадково закомітив файл `.env` з паролями та відправив його на публічний GitHub. Які наслідки це може мати і як уникнути такої ситуації в майбутньому?

30. Поясніть, навіщо потрібен файл `.github/workflows/ci.yml`. Що відбудеться, якщо один із кроків у ньому завершиться помилкою -чи оновиться сайт на Netlify?

ЛЕКЦІЯ 5. API · REST · HTTP · JSON · Node.js · CRUD

Мета лекції

Сформувати у студентів чітке розуміння шести ключових понять сучасної веб-розробки: API, REST, HTTP, JSON, Node.js та CRUD. Показати, як ці поняття пов'язані між собою і чому вони є основою будь-якої інформаційної системи, що розгортається у хмарі. Навчити студентів розпізнавати та виправляти типові помилки при розробці API.

План лекції

1. Що таке API, REST, HTTP, JSON, Node.js, CRUD і як це все пов'язано між собою.
2. Чому це вивчається у дисципліні «Розгортання інформаційних систем».
3. Що буде, якщо зробити все неправильно: реальні наслідки помилок.
4. API - що це таке: визначення, типи, приклади з реального життя.
5. REST та HTTP-методи: GET, POST, PUT, PATCH, DELETE і CRUD-операції.
6. HTTP-статус коди: що означає кожен клас відповідей.
7. JSON - мова, якою говорять API: синтаксис, типи, помилки.
8. API-ключі та безпека: де зберігати, що буде якщо не зберігати.
9. Асинхронний JavaScript: callback, Promise, async/await, Fetch API.
10. Express.js - пишемо власний API: Node.js, middleware, CRUD-сервер.
11. CORS - чому браузер блокує запити і як це виправити.
12. REST проти GraphQL, gRPC, WebSocket: коли що обирати.
13. Підсумок, глосарій та рекомендована література.

Ключові терміни лекції

API (Application Programming Interface) - набір правил, за якими одна програма звертається до іншої.

REST (Representational State Transfer) - архітектурний стиль побудови API поверх HTTP.

HTTP (HyperText Transfer Protocol) - протокол передачі даних, на якому побудований весь Інтернет.

JSON (JavaScript Object Notation) - текстовий формат для передачі даних між програмами.

Node.js - середовище для запуску JavaScript поза браузером, на сервері.

CRUD (Create, Read, Update, Delete) - чотири базові операції з даними у будь-якій системі.

Endpoint - конкретна адреса API, наприклад /api/todos - «кінцева точка», куди іде запит.

Middleware - функція, яка виконується між отриманням запиту і відправкою відповіді.

CORS (Cross-Origin Resource Sharing) - механізм дозволу запитів між різними сайтами.
async/await - сучасний синтаксис для зручної роботи з асинхронним кодом у JavaScript.

ЩО TAKE API, REST, HTTP, JSON, NODE.JS ТА CRUD - І ЯК ЦЕ ПАЗЛ СКЛАДАЄТЬСЯ РАЗОМ

Шість понять, які треба знати кожному - і чому

Уявіть, що вам потрібно побудувати сучасний сайт, наприклад інтернет-магазин. Покупець відкриває сторінку товару - і бачить фото, ціну, кількість на складі. Він натискає «Купити» - система перевіряє залишки, оформлює замовлення, надсилає email-підтвердження і передає дані кур'єрській службі. Все це відбувається за частки секунди. Як? Завдяки шести поняттям, які ви вивчаєте сьогодні.

API (Application Programming Interface) - це «офіціант» між вашим кодом і зовнішнім сервісом. Покупець не йде сам на «кухню» (сервер бази даних) - він робить замовлення через API, а API повертає результат. Без API кожен розробник мусив би писати власний код для кожного зовнішнього сервісу - оплати, карт, погоди, авторизації. З API - використовуємо готове, перевірене рішення за документацією.

REST (Representational State Transfer) - це набір правил «як правильно будувати API». Не технологія і не протокол, а архітектурний стиль. Придумав його Рой Філдінг у 2000 році у своїй докторській дисертації. Правила REST дозволяють будь-якому розробнику у будь-якій країні зрозуміти чужий API без пояснень - просто за назвами адрес та HTTP-методами.

HTTP (HyperText Transfer Protocol) - це протокол (мова спілкування) між браузером і сервером. Кожен раз, коли ви відкриваєте сторінку, браузер надсилає HTTP-запит. Кожен раз, коли ваш код звертається до API - теж HTTP-запит. HTTP визначає, яким методом звертатись (GET, POST, DELETE тощо) і які статус-коди повертати (200 - успіх, 404 - не знайдено, 500 - помилка сервера).

JSON (JavaScript Object Notation) - це формат, у якому дані «їдуть» по мережі між клієнтом і сервером. Уявіть: браузер надіслав HTTP-запит до API - і API повертає відповідь. У якому вигляді? У форматі JSON: зрозумілий текстовий рядок у фігурних дужках, який читається людиною і легко обробляється програмою. JSON переміг XML (попередній стандарт) завдяки простоті.

Node.js - це середовище для запуску JavaScript на сервері (поза браузером). Завдяки Node.js ви можете писати і клієнтський (браузерний) код, і серверний (API) однією мовою - JavaScript. Це величезна перевага: не потрібно вивчати окремо Python або PHP для сервера. Node.js + Express.js = ваш власний API-сервер за 30 рядків коду.

CRUD (Create, Read, Update, Delete) - це чотири базові операції з будь-якими даними у будь-якій системі. Будь-яка база даних, будь-яке API, будь-який сайт - по суті виконує ці чотири дії. Створення запису, читання запису, оновлення запису, видалення запису. REST API реалізує CRUD через HTTP-методи: POST (Create), GET (Read), PUT/PATCH (Update), DELETE (Delete).

Як ці шість понять пов'язані між собою

Ось повна картина того, що відбувається, коли ваш браузер запитує дані у API:

1. Браузер виконує JavaScript-код (написаний з використанням Node.js-середовища).
2. Код надсилає HTTP-запит методом GET, POST, PUT або DELETE - це CRUD-операція.
3. Запит адресований до конкретного endpoint API - адреси у стилі REST.
4. Якщо потрібно передати дані (наприклад, новий запис) - вони серіалізуються у JSON і кладуться у тіло запиту.
5. Сервер (Node.js + Express) отримує запит, обробляє, виконує операцію з базою даних.
6. Сервер повертає HTTP-відповідь зі статус-кодом (200, 201, 404 тощо) і JSON-тілом.
7. Браузер десеріалізує JSON назад в об'єкт і оновлює інтерфейс сторінки.

Ось чому не можна вивчити одне і пропустити інше: вони утворюють єдиний ланцюг. Якщо не розуміти HTTP - не зрозумієте чому запит повернув 403 замість 200. Якщо не розуміти JSON - не зможете надіслати дані на сервер. Якщо не розуміти REST - ваш API буде хаотичним і ніхто не зможе ним користуватись.

Поняття	Роль у ланцюгу	Без цього...
HTTP	Протокол - «мова» між браузером і сервером	Не буде зв'язку взагалі
REST	Правила іменування адрес і методів	API буде незрозумілим і хаотичним
CRUD	Чотири типи операцій з даними	Не зможете описати, що робить endpoint
JSON	Формат передачі даних у тілі запиту/відповіді	Не знатимете як серіалізувати/десеріалізувати
Node.js	Середовище для запуску серверного коду	Не зможете написати власний API
API	Контракт між клієнтом і сервером	Кожна частина системи буде ізольована

Чому це вивчається у дисципліні «Розгортання інформаційних систем»

«Розгортання інформаційних систем» - це не дисципліна про «як написати код». Це дисципліна про те, як зробити так, щоб ваш код запрацював у реальному світі і взаємодівав з іншими системами. А сучасний світ - це світ API.

Сьогодні практично будь-яка інформаційна система складається з кількох частин, які спілкуються між собою через API: фронтенд (сайт або мобільний додаток) надсилає запити до бекенду (сервера) через REST API; бекенд звертається до зовнішніх сервісів (оплата, пошта, SMS) через їхні API; мікросервіси всередині системи спілкуються між собою через внутрішні API. Коли ви розгортаєте інформаційну систему у хмарі (AWS, Azure, Google Cloud) - ви конфігуруєте все через API. Terraform, GitHub Actions, Netlify - всі вони керуються через API.

Конкретні зв'язки між темою і дисципліною:

- Автоматичне розгортання (CI/CD) використовує API GitHub для запуску деплою при push.
- Хмарні провайдери (AWS, Azure) надають API для створення серверів, баз даних, мереж.
- Моніторинг і логування - окремі мікросервіси, що приймають дані через API.
- Netlify Functions - це маленькі API-endpoint, що запускаються в хмарі без сервера.
- Мобільні додатки, сайти, IoT-пристрої - всі звертаються до хмарних систем через API.

ЩО БУДЕ, ЯКЩО ЗРОБИТИ НЕПРАВИЛЬНО: РЕАЛЬНІ НАСЛІДКИ ПОМИЛОК

Помилка 1 - Неправильне іменування endpoint-ів

Уявіть: ваш API має адреси /getTodos, /createTodo, /deleteTodoById?id=5. Це не REST - це «самопис». Інший розробник не зрозуміє логіку, документація буде незрозумілою, а інтеграція займе годинами замість хвилин.

Реальні наслідки: команда з 5 розробників витрачає по 2-3 зайвих години на тиждень тільки тому, що API-адреси названі непослідовно. За рік - 500+ годин витраченого часу. Для стартапу це може означати провалений дедлайн і втрату інвестицій.

Правильно: /todos (GET - всі, POST - новий), /todos/5 (GET - один, PATCH - оновити, DELETE - видалити). Один ресурс - одна адреса, дія визначається HTTP-методом.

Помилка 2 - Відкритий API-ключ у коді

Студент пише сервіс погоди. Отримує безкоштовний ключ OpenWeatherMap. Вставляє ключ прямо у JavaScript-файл: `const KEY = 'abc123'`. Завантажує на GitHub. Публічний репозиторій.

Реальні наслідки: автоматизовані боти сканують GitHub цілодобово. Ключ буде знайдений і проданий або використаний за лічені хвилини. OpenWeatherMap заблокує ваш ключ - сервіс зупиниться. Якщо це ключ AWS або Google Cloud - вам прийде рахунок на тисячі доларів. Реальний задокументований випадок 2022 року: студент заплатив \$47,000 за одну ніч роботи чужих серверів, які запустили зловмисники з його вкраденим AWS-ключем.

Правильно: усі секрети - у файлі .env, .env - у .gitignore. Читати через process.env.KEY_NAME. Ніколи не публікувати .env у репозиторій.

Помилка 3 - Синхронний код замість асинхронного

Новачок пише запит до API і не використовує await або .then(). Код виглядає так: `const data = fetch('/api/todos'); console.log(data);` - і виводить у консоль не дані, а об'єкт Promise.

Реальні наслідки: сторінка «зависає» на час очікування відповіді. При запиті, що займає 3 секунди, весь браузер перестає реагувати. Користувач думає, що сайт «завис», і закриває вкладку. За статистикою Google, якщо сторінка завантажується понад 3 секунди - 53% мобільних користувачів її покидають.

Правильно: завжди використовувати async/await або .then() при HTTP-запитах. Код повинен чекати відповідь асинхронно, не блокуючи інтерфейс.

Помилка 4 - Неправильні HTTP статус-коди у відповідях

Розробник написав API, який при будь-якій відповіді повертає статус 200 OK - навіть коли запис не знайдено або дані невалідні. Просто повертає JSON з полем error: 'not found' і статусом 200.

Реальні наслідки: весь моніторинг і логування стає марним - система вважає, що все добре (200 = успіх), хоча насправді половина запитів завершується помилкою. Клієнтський код не може автоматично визначити успіх чи провал без читання тіла відповіді. Інтеграція з іншими сервісами ламається. Налагодження займає у 3-5 разів більше часу.

Правильно: завжди використовувати правильні коди: 200 OK для успіху, 201 Created після POST, 400 Bad Request для невалідних даних, 401 Unauthorized якщо немає токена, 404 Not Found якщо запис не існує, 500 Internal Server Error при непередбаченій помилці сервера.

Помилка 5 - Відсутність валідації вхідних даних

API приймає запит POST /api/users з JSON-тілом і одразу зберігає дані у базу, не перевіряючи їх. Зловмисник надсилає замість email рядок з SQL-ін'єкцією або скрипт XSS.

Реальні наслідки: SQL-ін'єкція може знищити всю базу даних або дати доступ до всіх записів. XSS-атака дозволяє виконати шкідливий JavaScript у браузерах всіх відвідувачів сайту. За даними OWASP (Open Web Application Security Project), відсутність валідації - одна

з топ-3 причин зламів веб-додатків. Одна неперевірена точка вводу може коштувати компанії мільйони доларів та репутацію.

Правильно: завжди перевіряти вхідні дані: чи присутні обов'язкові поля, чи правильний тип даних, чи є email схожим на email, чи не перевищує рядок допустиму довжину. Повертати 400 Bad Request з описом помилки, якщо дані невалідні.

Помилка 6 - Ігнорування CORS

Студент написав API на Node.js, що запускається на порту 3000. Намагається звернутись до нього зі свого HTML-файлу (відкритого через file:// або з порту 8080). Браузер блокує запит: «CORS error».

Реальні наслідки: фронтенд повністю не може спілкуватись з бекендом. Розробник витрачає години на пошук «баги в коді», хоча API працює ідеально - просто відсутні CORS-заголовки. В продакшні неправильно налаштований CORS може або блокувати легітимних користувачів, або (якщо відкрити все підряд через cors()) - зробити API вразливим до CSRF-атак.

Правильно: у розробці використовувати `app.use(cors())` для дозволу всіх запитів. У продакшні - вказувати конкретний список дозволених origins через `cors({ origin: 'https://yourapp.com' })`.

API - ЩО ЦЕ TAKE

Просте визначення

API (Application Programming Interface - Прикладний Програмний Інтерфейс) - це набір правил, за якими одна програма звертається до іншої. Простіше кажучи: API - це домовленість між двома програмами про те, як вони спілкуються.

Ви не знаєте, як влаштований двигун автомобіля - але вмієте крутити кермо. Кермо і є «інтерфейс». Так само розробник не знає, як Google влаштував свої карти всередині - він просто використовує Google Maps API за документацією і отримує карту.

Аналогія: API як офіціант у ресторані. Ви (сайт або додаток) хочете отримати дані. Ви не йдете напругу «на кухню» (сервер) - там вас ніхто не чекає. Офіціант (API) приймає ваше замовлення у зрозумілому форматі, передає на кухню і повертає готовий результат. Ви не знаєте, що відбувається на кухні - і вам не треба знати. Це і є суть API.

Де ви вже бачили API?

API - не щось абстрактне. Ви зустрічаєтесь з ним щодня, не підозрюючи про це:

Що бачите на екрані	Який API працює за лаштунками	Хто його надає
---------------------	-------------------------------	----------------

Погода у телефоні	OpenWeatherMap API - повертає температуру і прогноз	OpenWeatherMap
«Увійти через Google»	Google OAuth API - перевіряє ваш акаунт	Google
Карта на будь-якому сайті	Google Maps API - відображає карту та маршрути	Google
Оплата карткою онлайн	Stripe або LiqPay API - проводить транзакцію	Stripe, LiqPay
Курс долара на сайті банку	API Національного банку України - повертає курси	НБУ (відкрите, безкоштовно)
Telegram-бот відповідає вам	Telegram Bot API - приймає і надсилає повідомлення	Telegram
Переклад тексту онлайн	DeepL або Google Translate API	DeepL, Google

Типи API за доступом

Доступ до API буває різним залежно від цілей і умов використання:

- **Відкритий (Public API)** - доступний усім. Наприклад, API НБУ або CoinGecko (курси криптовалют). Зазвичай безкоштовний до певної кількості запитів. Деякі не потребують навіть реєстрації - просто відкрийте URL у браузері.
- **Приватний (Private API)** - внутрішній, тільки для розробників самої компанії. Наприклад, API між різними частинами сайту Uber (карта, оплата, рейтинг - окремі сервіси, що спілкуються між собою через приватні API).
- **Партнерський (Partner API)** - для компаній, що підписали угоду. Google Maps для корпоративних клієнтів - більше можливостей за гроші (детальні звіти, вища пропускну здатність).

REST ТА HTTP-МЕТОДИ

Що таке REST

REST (Representational State Transfer - Передача репрезентативного стану) - це архітектурний стиль побудови API. Не протокол і не технологія, а набір правил «як правильно проектувати API». Придумав його Рой Філдінг у 2000 році у своїй докторській дисертації в Каліфорнійському університеті.

API, побудований за правилами REST, називають RESTful API. Головна ідея REST: кожен «ресурс» (користувач, замовлення, товар) має свою унікальну адресу, і ми звертаємось

до нього стандартними HTTP-методами. Сервер нічого не «пам'ятає» між запитами - кожен запит самодостатній і містить всю необхідну інформацію.

Чому REST переміг як стандарт? Простота: будь-який розробник розуміє `/users/5` без документації. Використання HTTP: не потрібен окремий протокол. Масштабованість: сервери без стану легко масштабуються. Сумісність: будь-яка мова програмування вміє HTTP.

HTTP-методи та CRUD-операції

HTTP - протокол, на якому побудований весь Інтернет. REST API використовує HTTP-методи для чотирьох базових операцій з даними: CRUD (Create, Read, Update, Delete) - створення, читання, оновлення, видалення. Це правило справедливе для БУДЬ-ЯКОЇ системи: чи це соціальна мережа, чи банківська система, чи онлайн-магазин - всі вони, по суті, роблять тільки CRUD.

HTTP-метод	CRUD-операція	Що робить	Приклад
GET	Read - читання	Отримати дані. Нічого не змінює на сервері. Безпечний і ідемпотентний - можна викликати скільки завгодно раз.	GET <code>/api/todos</code> - список завдань; GET <code>/api/todos/5</code> - одне завдання
POST	Create - створення	Створити новий запис. Кожен виклик - новий об'єкт. Не ідемпотентний - два однакових POST створять два записи.	POST <code>/api/todos</code> - нове завдання
PUT	Update - оновлення (повне)	Замінити існуючий запис цілком. Якщо поле не передано - воно стирається. Ідемпотентний.	PUT <code>/api/todos/5</code> - повна заміна завдання №5
PATCH	Update - оновлення (часткове)	Змінити тільки передані поля існуючого запису. Інші поля залишаються без змін.	PATCH <code>/api/todos/5</code> - змінити тільки статус завдання №5
DELETE	Delete - видалення	Видалити запис. Якщо його вже немає - поверне 404. Ідемпотентний - повторне видалення не дає нового результату.	DELETE <code>/api/todos/5</code> - видалити завдання №5

HTTP-статус коди

Коли сервер відповідає на запит, він завжди повертає тризначний код - статус. Це не просто формальність: статус-код - це машинозрозумілий спосіб сказати «все добре» або «щось пішло не так». Клієнтський код перевіряє статус і вирішує, що робити далі. Перша цифра показує клас відповіді:

Клас	Код	Назва	Простими словами
2xx - Успіх	200 OK	Запит виконано	Все добре, ось дані. Відповідь на GET, PATCH.
	201 Created	Ресурс створено	Новий запис успішно збережено. Повертається після POST.
	204 No Content	Тіло порожнє	Виконано, але відповідати нічим. Повертається після DELETE.
4xx - Помилка клієнта	400 Bad Request	Неправильний запит	Ви надіслали невалідні дані. Перевірте формат і обов'язкові поля.
	401 Unauthorized	Не авторизований	Потрібно увійти в систему або передати токен авторизації.
	403 Forbidden	Заборонено	Ви авторизовані, але прав на цю операцію немає.
	404 Not Found	Не знайдено	Такого ресурсу не існує. Перевірте ID або адресу endpoint.
	422 Unprocessable	Помилка валідації	Дані зрозумілі, але не проходять бізнес-логіку (наприклад, email вже зайнятий).
5xx - Помилка сервера	429 Too Many Requests	Забагато запитів	Перевищено Rate Limit. Зачекайте або зменште частоту запитів.
	500 Internal Server Error	Помилка сервера	Щось зламалося на сервері - це не ваша вина. Дивіться логи сервера.
	503 Service Unavailable	Сервіс недоступний	Сервер на технічному обслуговуванні або перевантажений.

Як легко запам'ятати: 2xx - все добре (двійка - добре); 4xx - ваша помилка (чотирма пальцями тичете в себе); 5xx - помилка сервера (сервер «упав»). Зайдіть на сайт [http.cat](http://cat) - там кожен статус-код проілюстрований фотографією кота з відповідним «настроєм». Смішно, але справді запам'ятовується!

Як виглядає HTTP-запит і відповідь

Ось реальний приклад. Ви хочете створити нове завдання. Браузер надсилає такий запит:

```
POST /api/todos HTTP/1.1
Host: localhost:3000
Content-Type: application/json

{
```

```
"title": "Вивчити REST API",  
"completed": false  
}
```

Сервер відповідає:

```
HTTP/1.1 201 Created  
Content-Type: application/json  
  
{  
  "id": 4,  
  "title": "Вивчити REST API",  
  "completed": false  
}
```

Як правильно називати адреси (endpoint-и)

Endpoint - це адреса, за якою живе ресурс. Є кілька простих правил, яких дотримуються всі досвідчені розробники:

Правило	Неправильно	Правильно
Іменники, не дієслова - HTTP-метод вже каже, що робити	/getTodos, /createUser, /deletePost	/todos, /users, /posts
Множина для колекцій	/todo, /user	/todos, /users
Один елемент - через ID	/getTodoById?id=5	/todos/5
Ієрархія через /	/userPosts, /commentsByPost	/users/5/posts, /posts/12/comments
Версіонування - щоб старе не ламалось при змінах	/todos (змінити - зламати клієнтів)	/api/v1/todos, /api/v2/todos
Малий регістр, дефіс замість underscore	/UserProfile, /user_profiles	/user-profiles

JSON - мова, якою говорять API

Що таке JSON

JSON (JavaScript Object Notation - Об'єктна нотація JavaScript) - текстовий формат для передачі даних між програмами. Попри назву, JSON не прив'язаний до JavaScript - його розуміє будь-яка мова програмування: Python, Java, C#, PHP, Go тощо. Виник у 2001 році як легша альтернатива XML. Сьогодні JSON - стандарт де-факто для REST API у всьому світі.

Чому JSON переміг XML? XML виглядає так: `<name>Олексій</name><age>21</age>`. JSON виглядає так: `{"name": "Олексій", "age": 21}`. JSON коротший, легше читається людиною і легше парситься комп'ютером. При передачі великих масивів даних через мережу менший розмір = менший трафік = швидша відповідь.

Ось JSON-об'єкт з усіма типами даних, які підтримує формат:

```
{
  "name": "Олексій",          // рядок - завжди у подвійних лапках
  "age": 21,                  // число - без лапок
  "isStudent": true,          // булеве - тільки true або false
  "middleName": null,         // null - значення відсутнє
  "courses": ["API", "Cloud"], // масив - у квадратних дужках
  "address": {                // вкладений об'єкт
    "city": "Одеса",
    "zip": "65000"
  }
}
```

Масив об'єктів - типова відповідь API

Коли ви запитуєте список (наприклад, усі завдання), API повертає масив JSON-об'єктів. Це найпоширеніша структура відповіді:

```
[
  { "id": 1, "title": "Вивчити Node.js", "completed": false },
  { "id": 2, "title": "Створити API", "completed": true },
  { "id": 3, "title": "Написати тести", "completed": false }
]
```

Найчастіші помилки у JSON

JSON має дуже суворий синтаксис. Одна помилка - і весь документ стає невалідним. Ось найпоширеніші помилки початківців:

- **Одинарні лапки замість подвійних:** `{'name': 'Олексій'}` - ПОМИЛКА. Правильно: `{"name": "Олексій"}`. В JSON тільки подвійні лапки!
- **Кома після останнього елементу:** `{"name": "Олексій"},` - ПОМИЛКА. В JSON так звана «trailing comma» (хвостова кома) заборонена. На відміну від JavaScript, де це дозволено.
- **Коментарі в JSON:** `{"name": "Олексій" // автор}` - ПОМИЛКА. JSON коментарі взагалі не підтримує. Для коментарів використовуйте окреме поле або переходьте на JSONC/JSON5.

- **Ключ без лапок:** {name: "Олексій"} - ПОМИЛКА. У JavaScript об'єктах так можна, а в JSON - ні. Кожен ключ обов'язково у подвійних лапках.
- **Undefined як значення:** {"value": undefined} - ПОМИЛКА. JSON не знає про undefined. Використовуйте null замість undefined.

Серіалізація та десеріалізація

Серіалізація - перетворення об'єкта у пам'яті на текстовий рядок JSON для відправки по мережі. Десеріалізація - зворотній процес: отриманий текст перетворюється на об'єкт, з яким можна працювати у коді. Обидва процеси відбуваються автоматично при використанні Fetch API та Express - але розробнику важливо розуміти, що «під капотом» відбувається перетворення:

```
const user = { name: 'Олексій', age: 21 };

// Серіалізація - об'єкт → рядок (для відправки по мережі)
const text = JSON.stringify(user);
// результат: '{"name":"Олексій","age":21}'

// Десеріалізація - рядок → об'єкт (після отримання відповіді)
const obj = JSON.parse(text);
// результат: { name: 'Олексій', age: 21 }

// Валідація JSON-рядка:
try {
  const data = JSON.parse(someString);
} catch(e) {
  console.error("Невалідний JSON:", e.message);
}
```

API-КЛЮЧІ ТА БЕЗПЕКА

Навіщо потрібен API-ключ?

API-ключ (API Key) - це унікальний рядок символів, який ідентифікує вашу програму перед API-провайдером. Уявіть його як перепустку: без неї на територію не пускають. Ключ потрібен провайдеру, щоб знати: хто надсилає запит; скільки запитів зробив цей клієнт; чи не перевищив він ліміт; чи не використовується ключ у зловмисних цілях.

Якщо ліміт запитів перевищено - ключ блокується або запити починають коштувати грошей. Якщо ключ вкрадений зломисниками - вони будуть відправляти запити від вашого імені і за ваш рахунок.

Де правильно зберігати API-ключ

```
// НЕПРАВИЛЬНО - ключ прямо в коді
// Якщо завантажити на GitHub - боти знайдуть його за хвилини!
const API_KEY = 'abc123def456secret';

// ПРАВИЛЬНО - ключ у файлі .env (він у .gitignore, не потрапляє в Git)

// Файл .env (НЕ завантажується на GitHub):
OPENWEATHER_API_KEY=abc123def456secret
STRIPE_SECRET_KEY=sk_live_...

// У файлі .gitignore обов'язково додайте:
.env
.env.local
.env.production

// У коді читаємо через process.env:
const apiKey = process.env.OPENWEATHER_API_KEY;
// Тепер ключа немає в коді, тільки у .env
```

Rate Limiting - ліміт запитів

Rate Limit - максимальна кількість запитів до API за одиницю часу. Перевищення ліміту → відповідь 429 Too Many Requests. Кожен провайдер встановлює свої ліміти:

API-провайдер	Безкоштовний ліміт	Практична порада
OpenWeatherMap	1,000 запитів/добу	Кешуйте відповідь - погода не міняється щохвилини. Зберігайте результат і оновлюйте раз на 10-15 хвилин.
GitHub REST API	60 запитів/год (без токена)	Завжди передавайте токен авторизації - тоді ліміт зростає до 5,000 запитів/год.
Google Maps API	\$200 кредиту/міс безкоштовно	Встановіть ліміт витрат у Google Cloud Console - захист від несподіваного рахунку.
API НБУ	Необмежено	Курс оновлюється раз на добу - частіше запитувати немає сенсу. Кешуйте на 24 години.

Telegram Bot API	30 повідомлень/сек на бот	При масових розсилках використовуйте черги повідомлень і затримки між запитами.
------------------	---------------------------	---

АСИНХРОННИЙ JAVASCRIPT

Чому «асинхронний»

JavaScript виконується в одному потоці - він може робити тільки одну справу за раз. Якщо HTTP-запит до API займає 2 секунди, синхронний код «заморозить» браузер на ці 2 секунди. Кнопки перестануть працювати, сторінка «зависне». Але ви бачите це щодня у зворотньому: завантажуєте файл і продовжуєте гортати стрічку - ніхто нікого не чекає. Це і є асинхронність.

Асинхронний код вирішує цю проблему: він запускає тривалу операцію (наприклад, HTTP-запит) і одразу «відпускає» браузер робити інші справи. Коли відповідь нарешті прийде - виконається потрібна функція з результатом. Браузер не блокується жодною миті.

Три покоління асинхронного коду

1. *Callback - старий підхід (до 2015 року)*

Callback - функція, яку передають як аргумент і викликають, коли операція завершиться. Основна проблема: при кількох послідовних запитах код стає нечитабельним - це називається «callback hell» або «піраміда смерті»:

```
getWeather('Kyiv', function(err, data) {
  if (err) return handleError(err);
  getForecast(data.id, function(err2, forecast) {
    if (err2) return handleError(err2);
    getAlerts(forecast.zone, function(err3, alerts) {
      // ще рівень... і ще... - важко читати і підтримувати!
    });
  });
});
```

2. *Promise - краще (ES6, 2015 рік)*

Promise (проміс) - об'єкт, що представляє результат, який з'явиться в майбутньому. Має три стани: pending (чекає виконання), fulfilled (виконано успішно), rejected (виконано з помилкою). Дозволяє «ланцюжок» .then() замість вкладених callbacks:

```
fetch('/api/weather?city=Kyiv')
```

```
.then(response => response.json())    // конвертуємо у JSON
.then(data => console.log(data))       // працюємо з даними
.catch(error => console.error(error)); // обробляємо помилки
```

3. async/await - сучасний підхід (ES2017, рекомендований)

async/await - найзручніший спосіб. Код виглядає майже як звичайний (синхронний), але не блокує браузер. Ключове слово await зупиняє виконання тільки поточної функції - решта браузера продовжує працювати нормально. Функція, що містить await, обов'язково повинна бути позначена async:

```
async function getWeather(city) {
  try {
    // await чекає відповіді, але не блокує браузер
    const response = await fetch(`/api/weather?city=${city}`);

    if (!response.ok) {
      throw new Error('Помилка: ' + response.status);
    }

    const data = await response.json(); // чекаємо парсингу JSON
    return data;

  } catch (error) {
    // обробляємо і мережеві помилки, і помилки статус-кодів
    console.error('Запит не вдался:', error.message);
  }
}

// Використання:
const weather = await getWeather('Одеса');
```

Fetch API - робимо HTTP-запити з браузера

Fetch API - вбудований у браузер інструмент для HTTP-запитів. Не потребує встановлення бібліотек. Повертає Promise, тому ідеально поєднується з async/await. Підтримується у всіх сучасних браузерах.

GET - отримати дані

```
// GET-запит - найпростіший: тільки URL
```

```
const response = await
fetch('https://bank.gov.ua/NBUStatService/v1/statdirectory/exchange?json');

if (!response.ok) {
  throw new Error(`HTTP помилка: ${response.status}`);
}

const rates = await response.json();
console.log(rates); // масив об'єктів з курсами валют
console.log(rates[0].rate); // наприклад: 41.5
```

POST - надіслати дані

```
// POST-запит - потрібно передати метод, заголовки і тіло
const response = await fetch('http://localhost:3000/api/todos', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json', // кажемо серверу: тіло - JSON
    'Authorization': `Bearer ${token}`, // токен авторизації (якщо є)
  },
  body: JSON.stringify({ // серіалізуємо об'єкт у JSON-рядок
    title: 'Нове завдання',
    completed: false
  })
});

const newTodo = await response.json();
console.log(newTodo.id); // ID новоствореного запису (201 Created)
```

EXPRESS.JS - ПИШЕМО ВЛАСНИЙ API

Node.js та Express.js - що це і навіщо?

Node.js - середовище для запуску JavaScript поза браузером, на сервері. Побудоване на рушії V8 - тому самому, що використовує Chrome. Завдяки Node.js ви можете писати серверний код мовою JavaScript, яку вже знаєте. Node.js ідеально підходить для API-серверів: він швидкий, асинхронний і добре масштабується при великій кількості паралельних запитів.

Express.js - мінімалістичний, але потужний фреймворк для Node.js, що спрощує створення веб-серверів та API. Аналогія: Node.js - це двигун автомобіля, а Express.js - зручна

панель управління з кермом, педалями і навігатором. Без Node.js Express не існує, але Node.js без Express - це писати все з нуля.

Що таке Middleware

Middleware (проміжний обробник) - функція, яка виконується між отриманням запиту і відправкою відповіді. Уявіть конвеєр на заводі: деталь (запит) проходить через кілька станцій (middleware) по черзі. Кожна станція може: обробити і передати далі (next()); або зупинити і відправити відповідь клієнту.

Приклади middleware в реальних проектах: express.json() - зчитує JSON з тіла запиту; cors() - додає CORS-заголовки; authMiddleware - перевіряє токен авторизації; loggerMiddleware - записує кожен запит у лог; errorHandler - обробляє всі помилки централізовано.

Повний CRUD API - покроковий приклад

Ось простий робочий сервер з усіма CRUD-операціями. Спочатку встановлення залежностей:

```
# Ініціалізуємо Node.js проект
npm init -y

# Встановлюємо Express і CORS
npm install express cors

# Запуск сервера
node server.js
```

Збережіть наступний код як server.js і запустіть командою node server.js:

```
const express = require('express');
const cors    = require('cors');
const app     = express();

app.use(express.json()); // middleware: читаємо JSON із тіла запиту
app.use(cors());         // middleware: дозволяємо запити з будь-якого origin

// Дані у пам'яті (у реальних проектах - база даних)
let todos = [
  { id: 1, title: 'Вивчити Node.js', completed: false },
  { id: 2, title: 'Створити API',      completed: false },
];
```

```
let nextId = 3;

// GET /api/todos - отримати всі завдання (Read)
app.get('/api/todos', (req, res) => {
  res.json(todos); // 200 OK автоматично
});

// GET /api/todos/:id - отримати одне завдання (Read)
app.get('/api/todos/:id', (req, res) => {
  const todo = todos.find(t => t.id === parseInt(req.params.id));
  if (!todo) return res.status(404).json({ error: 'Не знайдено' });
  res.json(todo);
});

// POST /api/todos - створити нове завдання (Create)
app.post('/api/todos', (req, res) => {
  const { title } = req.body;
  if (!title) return res.status(400).json({ error: 'Поле title обов\'язкове' });
  const newTodo = { id: nextId++, title, completed: false };
  todos.push(newTodo);
  res.status(201).json(newTodo); // 201 Created
});

// PATCH /api/todos/:id - оновити завдання частково (Update)
app.patch('/api/todos/:id', (req, res) => {
  const todo = todos.find(t => t.id === parseInt(req.params.id));
  if (!todo) return res.status(404).json({ error: 'Не знайдено' });
  if (req.body.title !== undefined) todo.title = req.body.title;
  if (req.body.completed !== undefined) todo.completed = req.body.completed;
  res.json(todo);
});

// DELETE /api/todos/:id - видалити завдання (Delete)
app.delete('/api/todos/:id', (req, res) => {
  const idx = todos.findIndex(t => t.id === parseInt(req.params.id));
  if (idx === -1) return res.status(404).json({ error: 'Не знайдено' });
  todos.splice(idx, 1);
  res.sendStatus(204); // 204 No Content - успішно, тіло порожнє
});

app.listen(3000, () => console.log('Сервер: http://localhost:3000'));
```

Як протестувати API без написання коду: завантажте Postman (postman.com) - безкоштовний інструмент. Оберіть метод (GET/POST/PATCH/DELETE), введіть URL (наприклад, http://localhost:3000/api/todos), для POST і PATCH - додайте JSON у Body → raw → JSON. Натисніть Send - побачите відповідь сервера з кодом і тілом.

CORS - ЧОМУ БРАУЗЕР БЛОКУЄ ЗАПИТИ

Same-Origin Policy - звідки береться помилка

Same-Origin Policy (політика одного походження) - правило безпеки браузера: JavaScript може робити запити тільки до того самого origin (походження), звідки завантажена сторінка. Origin = схема (http/https) + хост + порт. Навіть якщо відрізняється тільки порт - це вже різні origins.

Навіщо це правило? Захист від атак. Уявіть: ви зайшли на bank.ua і залишились авторизованими. Потім відкрили шкідливий сайт evil.com. Без Same-Origin Policy evil.com міг би від вашого імені надсилати запити до bank.ua, красти дані або переказувати гроші. Правило це забороняє.

Сторінка завантажена з	Запит до	Результат
http://localhost:3000	http://localhost:3000/api/todos	Дозволено - той самий origin
http://localhost:3000	http://localhost:4000/api/todos	CORS-помилка - різні порти (3000 і 4000)
https://myapp.com	https://api.myapp.com/todos	CORS-помилка - різний субдомен
https://myapp.com	http://myapp.com/api	CORS-помилка - https vs http
file:///index.html	http://localhost:3000/api	CORS-помилка - типово для студентів, що відкривають HTML-файл напряму!

Важливо: CORS - це захист БРАУЗЕРА, не сервера. Якщо ви робите запит через Postman або curl - CORS не діє. Тому: запит у Postman працює, а у браузері ні - проблема у відсутності CORS-заголовків на сервері, а не у вашому JavaScript-коді.

Вирішення CORS у Express

```
// npm install cors
```

```

const cors = require('cors');

// Для розробки - дозволити всі origins (зручно, але небезпечно в продакшні):
app.use(cors());

// Для продакшну - тільки конкретний дозволений сайт:
app.use(cors({
  origin: 'https://myapp.com',
  methods: ['GET', 'POST', 'PATCH', 'DELETE'],
  allowedHeaders: ['Content-Type', 'Authorization']
}));

// Для кількох дозволених origins:
const allowedOrigins = ['https://myapp.com', 'https://admin.myapp.com'];
app.use(cors({
  origin: (origin, callback) => {
    if (!origin || allowedOrigins.includes(origin)) {
      callback(null, true);
    } else {
      callback(new Error('CORS: origin не дозволений'));
    }
  }
}));

```

REST ПРОТИ ІНШИХ ПІДХОДІВ

Коли REST - не єдиний варіант

REST - найпоширеніший підхід для побудови API, але не єдиний. Для 95% студентських і більшості реальних проєктів REST цілком достатньо. Але корисно знати про альтернативи - коли і чому їх обирають:

Технологія	Що це	Коли підходить	Хто використовує
REST	Архітектурний стиль над HTTP + JSON. Найпростіший у вивченні та реалізації.	CRUD-операції, публічні API, більшість веб-додатків. Ваш вибір за замовчуванням.	GitHub API, Stripe, API НБУ, Netlify, практично всі публічні API

GraphQL	Мова запитів: клієнт сам обирає, які саме поля отримати у відповіді.	Коли різні клієнти (сайт і мобільний) потребують різні набори даних. Зменшує over-fetching.	GitHub v4, Shopify, Twitter v2, Facebook
gRPC	Швидка система викликів між серверами, бінарний формат (не JSON).	Коли мікросервіси всередині компанії спілкуються між собою і важлива максимальна швидкість.	Google Cloud, Netflix, Uber (внутрішні сервіси)
WebSocket	Постійне двостороннє з'єднання: сервер сам може надсилати дані клієнту.	Чат, онлайн-ігри, live-трансляції - де потрібна миттєва доставка даних без опитування.	Slack, Discord, онлайн-ігри, торгові термінали

Для більшості проектів - REST. Це стандарт галузі, найбільше туторіалів, найпростіше почати. GraphQL - якщо бачите, що різні клієнти отримують різні підмножини одних даних. WebSocket - якщо потрібен справжній реальний час (не «оновлення кожні 5 секунд», а миттєва доставка). gRPC - коли вже розбираєтесь у мікросервісах і вам важлива максимальна швидкість.

ГЛОСАРІЙ

Термін	Що означає	Запам'ятати як
API	Application Programming Interface - правила взаємодії між програмами	Офіціант між кухнею і клієнтом
REST	Архітектурний стиль побудови API поверх HTTP	Набір правил «як правильно»
HTTP	Протокол передачі даних у вебі	Мова запитів між браузером і сервером
CRUD	Create, Read, Update, Delete - 4 базові операції з даними	POST / GET / PUT+PATCH / DELETE
Endpoint	Конкретна адреса API, наприклад /api/todos	«Кінцева точка» - куди іде запит
JSON	Текстовий формат обміну даними	Ключ-значення у подвійних лапках
Серіалізація	Перетворення об'єкта на рядок для передачі	JSON.stringify(obj)

Десеріалізація	Перетворення рядка назад на об'єкт	JSON.parse(str)
API Key	Унікальний ключ-перепустка до API	Зберігай у .env, ніколи не в коді!
Rate Limit	Ліміт кількості запитів за час	429 = «стоп, забагато запитів»
Middleware	Функція, що виконується між запитом і відповіддю	Станція на конвеєрі обробки
CORS	Механізм дозволу запитів між різними сайтами	Браузерний охоронець
Same-Origin Policy	Браузер дозволяє запити тільки до того ж схема+хост+порт	Інший порт = чужий origin
async/await	Синтаксис для зручної роботи з асинхронним кодом	Чекає відповідь, не заморожуючи браузер
Promise	Об'єкт - «обіцянка» майбутнього результату	pending → fulfilled або rejected
Fetch API	Вбудований браузерний інструмент для HTTP-запитів	fetch(url) → Promise → .json()
Node.js	Середовище для запуску JS на сервері (поза браузером)	JS + сервер = Node.js
Express.js	Популярний фреймворк для написання API на Node.js	npm install express

ПІДСУМОК ЛЕКЦІЇ

Ця лекція охопила шість фундаментальних понять, які є основою будь-якої сучасної інформаційної системи. Вони не є ізольованими темами - разом вони утворюють єдиний ланцюг: HTTP-запит → REST-адреса → JSON-тіло → Node.js-обробник → CRUD-операція → API-відповідь.

#	Ключова думка	Практичний висновок
1	API - це контракт між клієнтом і сервером. Без API частини системи не можуть спілкуватись.	Перш ніж писати код - визначте: які endpoint-и потрібні, які методи, які дані.
2	REST + HTTP-методи = CRUD. GET читає, POST створює, PATCH/PUT оновлює, DELETE видаляє.	Запам'ятайте відповідність CRUD ↔ HTTP-методи. Це використовується щодня.
3	Статус-коди - машинозрозуміла мова результату. 2xx = добре, 4xx = ваша помилка, 5xx = сервер.	Завжди повертайте правильний статус-код. Це основа моніторингу і налагодження.

4	JSON - стандарт де-факто для передачі даних. Суворий синтаксис, тільки подвійні лапки.	Валідуйте JSON перед відправкою. Використовуйте <code>JSON.stringify()</code> і <code>JSON.parse()</code> .
5	API-ключі - у <code>.env</code> , ніколи у коді. Витік ключа = фінансові втрати і зупинка сервісу.	Перевіряйте <code>.gitignore</code> перед кожним <code>git push</code> . Одна помилка = тисячі доларів збитку.
6	<code>async/await</code> - правильний спосіб роботи з HTTP-запитами. Без нього браузер «зависає».	Кожна функція з <code>fetch()</code> повинна бути <code>async</code> . Завжди використовуйте <code>try/catch</code> .
7	CORS - це захист браузера. Якщо в Postman працює, а в браузері ні - забули CORS на сервері.	Додайте <code>app.use(cors())</code> у Express. У продакшні - вкажіть конкретний <code>origin</code> .

Якщо щось незрозуміло - спробуйте самотійно: відкрийте у браузері bank.gov.ua/NBUStatService/v1/statdirectory/exchange?json і подивіться на реальний JSON від API НБУ. Це безкоштовно, без реєстрації і найкращий спосіб побачити все, що вивчили сьогодні.

КОРИСНІ РЕСУРСИ

Спробуйте самотійно - без реєстрації

- **API НБУ (курси валют):** bank.gov.ua/NBUStatService/v1/statdirectory/exchange?json - просто відкрийте у браузері і побачите JSON з курсами валют. Перший контакт з реальним API!
- **JSONPlaceholder:** jsonplaceholder.typicode.com - фейковий API для практики. Є `/posts`, `/users`, `/todos`. Ідеально для перших спроб з `fetch`.
- **HTTP Cats:** http.cat - HTTP статус-коди з котиками. Смішно і корисно для запам'ятовування.
- **JSON Validator:** jsonlint.com - перевірте валідність вашого JSON онлайн. Показує точне місце помилки.

Документація

- **MDN - Fetch API:** developer.mozilla.org/uk/docs/Web/API/Fetch_API - є українською мовою.
- **Express.js:** expressjs.com - офіційна документація.
- **Node.js:** nodejs.org/uk - є українською.
- **Список публічних API для практики:** github.com/public-apis/public-apis - сотні безкоштовних API.

Відео для початківців

- **Fireship - REST API у 100 секунд:** youtu.be/-MTSQjw5DrM - коротко і зрозуміло.

- **Web Dev Simplified - async/await:** youtu.be/V_Kr9OSfDeU - найкраще пояснення для початківців.
- **Traversy Media - Express Crash Course:** youtu.be/L72fhGm1tfE.
- **freeCodeCamp - APIs for Beginners:** youtu.be/GZvSYJDk-us - 4-годинний безкоштовний курс.

КОНТРОЛЬНІ ПИТАННЯ

Блок 1. Git -система контролю версій

8. Що таке Git і яку проблему він вирішує? Наведіть конкретний приклад ситуації, де відсутність Git призводить до втрати роботи.
9. Поясніть різницю між поняттями *repository*, *commit* і *branch*. Наведіть приклад використання кожного.
10. Що таке гілка (branch) і навіщо вона потрібна? Опишіть сценарій, коли робота без гілок призведе до проблем.
11. Назвіть та поясніть не менше п'яти базових команд Git, якими розробник користується щодня.
12. Що таке Pull Request? Опишіть повний цикл роботи з PR від відкриття до злиття у main.
13. Що таке файл .gitignore і яка його роль? Які три категорії файлів обов'язково потрібно додавати до .gitignore?
14. Що таке «Conventional Commits» і які префікси використовуються для позначення типів змін? Чому важливо писати змістовні повідомлення до комітів?

Блок 2. GitHub -від реєстрації до першого репозиторію

13. Чим відрізняється Git від GitHub? Поясніть за допомогою аналогії з відомими програмами.
14. Перерахуйте не менше п'яти реальних наслідків, до яких призводить незнання або ігнорування GitHub у роботі розробника.
15. Що таке GitHub Pages? Які типи сайтів він підтримує, а які -ні?
16. Які кроки необхідно виконати, щоб прив'язати локальну папку проекту до репозиторію на GitHub?
17. Що таке GitHub Actions? Для чого він використовується та скільки хвилин роботи надається безкоштовно?

Блок 3. GitHub Student Developer Pack

16. Що таке GitHub Student Developer Pack і яку загальну вартість інструментів він надає безкоштовно?

17. Назвіть не менше чотирьох складових Student Pack та вкажіть їхню звичайну вартість.

18. Яка умова є найшвидшим способом підтвердити статус студента при реєстрації у Student Pack?

Блок 4. Тарифи GitHub

18. Порівняйте плани GitHub Free та GitHub Pro: чим вони відрізняються за функціями GitHub Actions та GitHub Pages?

19. Чи достатньо плану GitHub Free для виконання завдань першого курсу? Обґрунтуйте відповідь.

Блок 5. Netlify -хостинг для сайту

23. Що таке Netlify і яка його ключова ідея щодо автоматизації публікації сайтів?

24. Що таке CDN і чому завдяки йому сайти на Netlify завантажуються швидше, ніж на звичайному хостингу?

25. Опишіть покроково, як задеплоїти сайт на Netlify через підключення до GitHub.

26. Що таке Netlify Forms і яку перевагу вона дає студенту при створенні портфоліо?

27. Що таке Netlify Functions і для яких завдань вони використовуються?

Блок 6. Тарифи Netlify

25. Які обмеження має безкоштовний план Netlify (Starter) за трафіком, хвилинами збірки та формами?

26. Чи достатньо безкоштовного плану Netlify для навчального проекту першого курсу? Обґрунтуйте.

Блок 7. CI/CD -автоматичний деплой

29. Розшифруйте аббревіатуру CI/CD та поясніть, що означає кожна з частин.

30. Порівняйте ручний деплой (без CI/CD) і автоматичний (з Netlify + GitHub). У чому переваги автоматизації?

31. Що означають статуси деплою на Netlify: *Building*, *Published*, *Failed*, *Skipped*? Що слід робити при статусі *Failed*?

32. Назвіть чотири найпоширеніші причини невдалого (Failed) деплою на Netlify.

Питання підвищеної складності

31. Студент випадково закомітив файл `.env` з паролями та відправив його на публічний GitHub. Які наслідки це може мати і як уникнути такої ситуації в майбутньому?

32. Поясніть, навіщо потрібен файл `.github/workflows/ci.yml`. Що відбудеться, якщо один із кроків у ньому завершиться помилкою -чи оновиться сайт на Netlify?

Лекція 6. КОНТЕЙНЕРИЗАЦІЯ ДОДАТКІВ З DOCKER. СТВОРЕННЯ ТА РОЗГОРТАННЯ БАГАТОКОМПОНЕНТНИХ СИСТЕМ

Мета лекції

Дисципліна «Розгортання інформаційних систем» вчить вас не просто написати програму, а довести її до стану, коли нею можуть користуватись реальні користувачі в реальних умовах. Docker є центральним інструментом цього процесу з п'яти причин.

План лекції

1. Що таке Docker, Dockerfile, Docker Compose, Image, Container, Volume, Network - детальне пояснення та взаємозв'язок понять
2. Чому це вивчаємо в дисципліні «Розгортання інформаційних систем»
3. Що буде, якщо зробити неправильно: конкретні наслідки та помилки
4. Проблема «Works on my machine» та що таке контейнеризація
5. Віртуальні машини (VM) vs Docker: порівняння та відмінності
6. Docker Image та Docker Container: будова та принцип роботи
7. Dockerfile - рецепт побудови образу (детальний розбір кожної інструкції)
8. Docker CLI: основні команди для роботи з контейнерами
9. Docker Compose: оркестрація багатокомпонентних систем
10. Docker Volumes: збереження даних між перезапусками
11. Мережі в Docker: як контейнери спілкуються між собою
12. Рекомендована література

Що таке Docker, Dockerfile, Docker Compose, Image, Container, Volume, Network - детальне пояснення та взаємозв'язок

Чому ви зобов'язані знати ці поняття як майбутній фахівець? Уявіть, що ви влаштувались на першу роботу як junior-розробник. Перше, що вам скажуть: «Знімай репозиторій, запускай docker-compose up - і все працює.»

Якщо ви не розумієте, що таке Docker, Dockerfile, Volume, Network - ви не зможете навіть запустити проєкт. Це не перебільшення: Docker сьогодні є обов'язковою навичкою для 90% вакансій у веб-розробці, DevOps та хмарних технологіях.

Docker - що це і для чого

Docker - це платформа для контейнеризації програмного забезпечення. Простіше кажучи: Docker дозволяє «запакувати» ваш додаток разом із усім необхідним для його роботи (бібліотеками, конфігурацією, середовищем виконання) в єдиний ізольований пакет - контейнер. Цей контейнер запускається однаково на будь-якій машині, де встановлений

Docker: на вашому ноутбукі з Windows, MacBook викладача, сервері в хмарі AWS чи Google Cloud.

Уявіть конвеєр на заводі: деталі збираються в єдиний готовий продукт, упакований так, щоб його можна було відправити будь-куди і там він виглядав та працював так само. Docker робить те саме з програмним забезпеченням.

Коли з'явився Docker? У 2013 році компанія dotCloud (пізніше перейменована на Docker Inc.) відкрила свій внутрішній інструмент. За кілька років Docker став галузевим стандартом. Сьогодні понад 90% нових проєктів у хмарі розгортаються у контейнерах.

Dockerfile - рецепт для створення образу

Dockerfile - це звичайний текстовий файл (без розширення, з великої літери D), який містить покрокову інструкцію для побудови Docker-образу вашого додатку. Думайте про нього як про кулінарний рецепт: рецепт сам по собі - не їжа, але дотримуючись його, будь-хто приготує ідентичну страву.

Dockerfile вирішує одну критичну проблему: замість того, щоб писати довгі інструкції «як налаштувати середовище» (які ніхто не читає або читає не уважно), ви пишете машинозрозумілий рецепт, який Docker виконає точно і автоматично - незалежно від машини.

Проста аналогія: Dockerfile - це ніби рецепт борщу. Образ (Image) - це зварений борщ у каструлі. Контейнер (Container) - це тарілка борщу для конкретного гостя. З одного рецепту можна приготувати скільки завгодно порцій.

Image (образ) - заморожений шаблон

Docker Image (образ Docker) - це незмінний (immutable) шаблон для запуску контейнерів. Образ містить файлову систему з усім необхідним для роботи додатку: операційну систему (мінімальний набір), бібліотеки, ваш код та інструкцію запуску.

Технічно образ складається із шарів (layers). Кожна інструкція у Dockerfile створює новий незмінний шар. Шари кешуються: якщо ви змінили лише свій код, Docker не перезавантажує всі попередні шари - він бере їх із кешу. Це значно прискорює збирання образів.

Аналогія з ООП: Image (образ) - це як клас (Class) у об'єктно-орієнтованому програмуванні. Container (контейнер) - це екземпляр (Instance) цього класу. З одного Image можна запустити десятки однакових Container.

Container (контейнер) - запущений екземпляр

Docker Container (контейнер) - це запущений екземпляр Image. Технічно це один або кілька Linux-процесів, ізольованих від решти системи. Ізоляція досягається за допомогою двох механізмів ядра Linux:

- **Namespaces (простори імен)** - ізолюють процеси, файлову систему, мережу. Контейнер «бачить» лише свої процеси, немов він один на машині.
- **cgroups (control groups)** - обмежують ресурси (CPU, RAM). Наприклад: цей контейнер може використати максимум 512 МБ RAM.

Критично важливо: коли контейнер зупиняється або видаляється - всі зміни файлової системи (записи в БД, завантажені файли) зникають. Саме тому потрібні Volume.

Volume (том) - постійне сховище даних

Docker Volume (том) - це механізм збереження даних поза контейнером. Volume - це директорія на хост-машині (або в хмарному сховищі), підключена до контейнера. Дані у Volume зберігаються незалежно від lifecycle (циклу існування) контейнера: навіть якщо контейнер видалено - дані залишаються.

Для чого використовується: бази даних (PostgreSQL, MySQL, MongoDB), завантажені файли користувачів, логи, SSL-сертифікати. Без Volume при кожному docker-compose down ваша база даних очищується повністю!

Існує два типи: Named Volume (Docker керує розташуванням, рекомендується для БД) та Bind Mount (ви вказуєте конкретну директорію, зручно при розробці для hot-reload).

Network (мережа) - як контейнери спілкуються

Docker Network (мережа) - це віртуальна мережа між контейнерами. Без мережі контейнери ізольовані і не можуть спілкуватись між собою. Docker Compose автоматично створює bridge-мережу для всіх сервісів у файлі docker-compose.yml.

Ключова перевага: Docker Compose автоматично надає кожному сервісу DNS-ім'я, рівне імені сервісу. Якщо сервіс БД називається database - з веб-сервера до нього можна звертатись просто: postgresql://database:5432/myapp. Не потрібно знати IP-адресу!

Ізоляція мереж: різні docker-compose проєкти отримують різні мережі і не «бачать» одне одного. Це важливо для безпеки при розгортанні кількох проєктів на одному сервері.

Як усі поняття пов'язані між собою - загальна схема

Ланцюжок від коду до запущеного додатку:

Ваш код + залежності → Dockerfile (рецепт) → docker build → Image (образ) → docker run / docker-compose up → Container (запущений процес).

Volume підключається до Container для збереження даних. Network з'єднує кілька Container між собою. Docker Compose описує всю систему (усі Container, Volume, Network) в одному файлі docker-compose.yml.

Поняття	Що це	Аналогія	Для чого потрібно
Dockerfile	Текстовий файл-рецепт для збирання Image	Рецепт страви	Описує середовище: ОС, бібліотеки, код
Image	Незмінний шаблон середовища, складається з шарів	Клас в ООП / готова страва в каструлі	Шаблон для запуску контейнерів
Container	Запущений екземпляр Image, ізольований процес	Об'єкт класу / тарілка борщу для гостя	Реально виконує ваш додаток
Volume	Директорія поза контейнером для постійних даних	Зовнішній жорсткий диск для хмари	Зберігає дані БД, файли між перезапусками
Network	Віртуальна мережа між контейнерами з DNS	Внутрішня LAN-мережа офісу	Дозволяє web-серверу знайти базу даних
Docker Compose	Інструмент опису та запуску multi-container систем	Диригент оркестру	Керує всіма сервісами однією командою

Чому Docker вивчається в дисципліні «Розгортання інформаційних систем»

Дисципліна «Розгортання інформаційних систем» вчить вас не просто написати програму, а довести її до стану, коли нею можуть користуватись реальні користувачі в реальних умовах. Docker є центральним інструментом цього процесу з п'яти причин.

Причина 1: Docker - стандарт галузі. Понад 90% сучасних хмарних платформ (AWS, Google Cloud, Azure, Heroku, DigitalOcean) очікують, що ваш додаток упакований у Docker-контейнер. Без Docker ви не зможете розгорнути проект на жодній сучасній платформі.

Причина 2: Відтворюване середовище. Розгортання - це про те, щоб те, що «бо про це» на вашому ноутбукі, так само працювало на сервері, у колеги, у CI/CD системі. Dockerfile гарантує ідентичне середовище скрізь.

Причина 3: CI/CD пайплайни використовують Docker. GitHub Actions, GitLab CI, Jenkins - всі вони запускають ваші тести та збирають код всередині Docker-контейнерів. Без Docker ви не можете налаштувати автоматичне тестування і розгортання.

Причина 4: Мікросервісна архітектура. Сучасні системи складаються з десятків незалежних сервісів. Docker Compose дозволяє описати всю систему (Node.js API + PostgreSQL + Redis + Nginx) в одному файлі і запустити одною командою.

Причина 5: Безпека та ізоляція. Кожен сервіс у своєму контейнері з окремою мережею та обмеженнями ресурсів. Якщо один сервіс атакований - він не може шкодити іншим.

Що буде, якщо зробити неправильно: конкретні наслідки

Увага! Цей розділ - про реальні наслідки. Кожна помилка нижче - це не гіпотетична ситуація, а типова проблема, яка трапляється з розробниками щодня в реальних компаніях.

Помилка 1: Не використовувати Docker (покладатись на ручне налаштування)

Що відбувається: Ваш додаток написаний під Node.js 20. На сервері - Node.js 18. Деякі API несумісні. Додаток падає з незрозумілими помилками. Витрати: 2–3 години на діагностику, ще 2 години на ручне оновлення сервера, ризик зламати інші проєкти на тому ж сервері.

Реальний кейс: Команда з 5 розробників без Docker витрачала ~10 годин на тиждень на «у мене працює, у тебе - ні» проблеми. Це 500+ годин на рік - майже 3 місяці повноцінної роботи витрачено на те, що Docker вирішує повністю.

Помилка 2: Не використовувати Volume для бази даних

Що відбувається: Ви розробляєте проєкт, заносите тестові дані в базу PostgreSQL. Наступного дня запускаєте `docker-compose down && docker-compose up`. Вся база даних порожня. Всі ваші дані зникли.

Чому так: При видаленні контейнера видаляється і writable layer - шар файлової системи, де PostgreSQL зберігав дані. Без Named Volume - дані існують лише стільки, скільки живе контейнер.

Наслідки у продакшні: Якщо це трапляється на робочому сервері - це катастрофа: втрата даних користувачів, можливий простій бізнесу, штрафи за GDPR (Загальний регламент ЄС про захист даних). Компанії платили мільйони за такі інциденти.

Помилка 3: Зберігати секрети (паролі, API-ключі) у Dockerfile або коді

Що відбувається: Ви пишете у Dockerfile: `ENV DB_PASSWORD=myspassword123`. Або у коді: `const apiKey = 'sk-1234abcd...'`. Публікуєте на GitHub. Через 10 хвилин боти автоматично знаходять ключ і починають використовувати вашу хмарну інфраструктуру.

Реальний інцидент (2022 рік): Розробник випадково опублікував AWS-ключ на GitHub. За 3 години до виявлення зловмисники запустили 500 GPU-екземплярів для майнінгу криптовалюти. Рахунок - \$47,000 (майже 2 млн гривень). AWS частково відшкодував збитки, але не завжди.

Правило: Секрети зберігаються виключно у файлі .env, який внесений до .gitignore та .dockerignore. Ніколи у коді, ніколи у Dockerfile!

Помилка 4: Не ізолювати мережі контейнерів

Що відбувається: Всі контейнери в одній мережі за замовчуванням. Ваша база даних доступна з будь-якого контейнера, включаючи потенційно скомпрометований. Зловмисник, отримавши доступ до одного контейнера, може атакувати всі інші.

Правильний підхід: Окрема мережа frontend (Nginx + React), окрема мережа backend (Node.js API + PostgreSQL). База даних недоступна з Nginx напряму - тільки через API.

Помилка 5: Запускати контейнери від root-користувача

Що відбувається: За замовчуванням процеси в контейнері запускаються від root (суперкористувача). Якщо зловмисник знайде вразливість у вашому додатку і «вирветься» з ізоляції контейнера - він отримає root-доступ до всього сервера. Це критична уразливість безпеки.

Рішення: Завжди додавайте USER node (або USER appuser) перед CMD у Dockerfile після встановлення всього необхідного.

Помилка 6: Не перевіряти стан сервісів при запуску (depends_on без healthcheck)

Що відбувається: У docker-compose.yml вказано depends_on: database. Ваш Node.js API запускається, бачить, що контейнер з PostgreSQL стартував, і намагається підключитись. Але PostgreSQL ще ініціалізується всередині (це займає 5–10 секунд). API отримує помилку підключення і аварійно завершується.

Правильне рішення: depends_on з condition: service_healthy та healthcheck у сервісі бази даних. Тоді API чекатиме, поки PostgreSQL не буде реально готовий приймати підключення.

Зведена таблиця: помилки та їхні наслідки

Помилка	Наслідок	Рішення
Без Docker, ручне налаштування	500+ годин/рік на "у мене працює"	Dockerfile для кожного проєкту
Без Volume для БД	Всі дані зникають при docker-compose down	Named Volume у docker-compose.yml
Секрети у Dockerfile / коді	\$47,000 рахунок AWS (реальний кейс 2022)	.env у .gitignore, .dockerignore
Без ізоляції мереж	БД доступна з усіх контейнерів	Окремі networks у Compose

Контейнери від root	Root-доступ до сервера при зламі	USER node у Dockerfile
depends_on без healthcheck	API падає при старті, БД ще не готова	condition: service_healthy

Проблема «Works on my machine» та що таке контейнеризація

Уявіть таку ситуацію. Ви - студент першого курсу. Написали веб-додаток, він чудово запускається на вашому ноутбукі з Windows 11. Показуєте викладачу - а у нього macOS, інша версія Node.js, бібліотеки встановлені в іншу директорію. Dodatok не запускається. Або гірший варіант: ви здаєте проєкт роботодавцю на Linux-сервер - і там теж не працює.

Це явище настільки поширене в індустрії, що отримало власну назву: **«Works on my machine»** (укр. «Працює на моїй машині»). Причини зазвичай банальні:

- **Різні версії мови програмування:** Python 3.9 на сервері та Python 3.12 локально - між ними є несумісні зміни API.
- **Різні версії бібліотек:** ваш package.json містить "express": "^4.18", але на сервері встановлена версія 4.16.
- **Різні операційні системи:** на Windows шляхи пишуться через \ (зворотний слеш), на Linux/macOS - через / (прямий слеш).
- **Різні змінні оточення:** у вас локально є файл .env з налаштуваннями, на сервері - ні.
- **Різні системні бібліотеки:** деякі npm-пакети (bcrypt, sharp) компілюються під конкретну ОС і не переносяться.

Контейнеризація (Containerization) - підхід до пакування програмного забезпечення, при якому додаток разом із усіма своїми залежностями упаковується в єдиний ізольований пакет - контейнер. Ключова ідея: **«Build once, run anywhere»** - зібрати один раз, запускати будь-де.

Віртуальні машини (VM) vs Docker: порівняння

Що таке Virtual Machine (VM)? До Docker для ізоляції середовищ використовували VM. Гіпервізор (Hypervisor) - спеціальна програма - емулює повноцінний комп'ютер із власним процесором та пам'яттю. На ньому встановлюється повна Guest OS (гостьова ОС). Найбільший недолік: кожна VM несе повну копію ОС (~1.5 ГБ для Ubuntu). Запуск займає 1–5 хвилин.

Як Docker інший? Docker не емулює повний комп'ютер. Він використовує механізми ядра Linux (namespaces та cgroups), які вже існують у Host OS. Контейнер - це просто ізольований Linux-процес. Запуск займає менше секунди, розмір образу - лічені мегабайти.

Критерій	VM (Віртуальна машина)	Docker (Контейнер)
Час запуску	1–5 хвилин (завантаження повної ОС)	Менше 1 секунди
Розмір образу	1–20 ГБ (повна ОС у кожній VM)	10–200 МБ (тільки потрібне)
Споживання RAM	Кілька ГБ (ОС + додаток)	Мегабайти (тільки додаток)
Ізоляція	Повна - окреме ядро ОС	Часткова - спільне ядро Linux
Портабельність	Обмежена (залежить від гіпервізора)	Однаково на будь-якому Linux/Mac/Win
Підходить для	Різні ОС на одній машині, повна ізоляція	Мікросервіси, CI/CD, dev-середовища

Важливо: VM і Docker - не конкуренти, а доповнення. Типова хмарна архітектура: хмарна VM (AWS EC2) → на ній Docker Engine → всередині Docker кілька контейнерів із мікросервісами.

Docker Image та Container: будова та принцип роботи

Будова Image - шари (layers)

Image технічно складається з шарів. Кожна інструкція у Dockerfile створює новий незмінний шар. Шари кешуються: якщо змінити тільки код (верхній шар) - Docker перезбирає лише його, решта береться з кешу. Це значно пришвидшує CI/CD.

Приклад шарів для Node.js-додатку:

```
FROM node:18-alpine    <- Шар 1: базова ОС + Node.js (~50 МБ)
WORKDIR /app           <- Шар 2: створення директорії (< 1 КБ)
COPY package.json ./    <- Шар 3: файл залежностей (~1 КБ)
RUN npm install         <- Шар 4: встановлені бібліотеки (~20 МБ)
COPY . .               <- Шар 5: ваш код (~100 КБ)
CMD ["node", "app.js"]  <- Шар 6: команда запуску (< 1 КБ)
```

При зміні app.js і повторному docker build - шари 1–4 беруться з кешу (миттєво), перебудовуються лише 5 і 6. Саме тому у Dockerfile package.json копіюють ПЕРЕД кодом: npm install виконується рідко (тільки при змінах залежностей).

Docker Hub - «GitHub для образів»

Docker Registry (реєстр образів) - сховище для образів. Найвідоміший публічний реєстр - Docker Hub (hub.docker.com) - містить мільйони образів. Офіційні образи (з позначкою Docker Official Image) підтримуються командами розробників: образ postgres - командою PostgreSQL, node - командою Node.js.

Команда `docker pull postgres:15` автоматично завантажує образ PostgreSQL версії 15 з Docker Hub. Окрім Docker Hub існують приватні реєстри: AWS ECR, GCR (Google), GitLab Container Registry.

Dockerfile - детальний розбір інструкцій

Основні інструкції Dockerfile

FROM - базовий образ. Перша і обов'язкова інструкція. Вказує, на основі якого образу будуватиметься ваш. Завжди вказуйте конкретну версію (node:18), а не latest - тег latest може змінитись і зламати ваш образ.

```
FROM node:18-alpine # легкий Alpine Linux + Node.js 18
FROM python:3.12-slim # Python 3.12 на Debian slim
```

WORKDIR - робоча директорія. Встановлює директорію всередині контейнера. Якщо не існує - Docker створює її. Без WORKDIR файли потрапляли б у корінь (/), змішуючись із системними.

```
WORKDIR /app
```

COPY - копіювання файлів. Копіює файли з вашого комп'ютера у файлову систему образу.

```
COPY package*.json ./ # копіювати спочатку залежності (для кешу)
COPY . . # потім увесь код
```

RUN - виконання команд. Виконує команду під час збирання та зберігає результат як новий шар.

```
RUN npm install
RUN apk add --no-cache curl git # Alpine
```

ENV - змінні оточення. Встановлює змінні, доступні під час збирання та роботи контейнера. Увага: ніколи не зберігайте паролі та API-ключі у ENV у Dockerfile!

```
ENV NODE_ENV=production
ENV PORT=3000
```

EXPOSE - документація порту. Документує, що контейнер слухає певний порт. Важливо: EXPOSE не відкриває порт назовні! Це тільки метадані. Порт відкривається при `docker run -p 8080:3000`.

```
EXPOSE 3000
```

CMD - команда запуску. Визначає команду при запуску контейнера. Може бути замінена при docker run.

```
CMD ["node", "app.js"] # exec form (рекомендований)
```

USER - запуск від непривілейованого користувача. Критично для безпеки!

```
USER node # перемикнутись з root на node-користувача
```

Повний Dockerfile для Node.js-додатку

```
# Базовий образ: Node.js 18 + Alpine Linux
FROM node:18-alpine

# Робоча директорія
WORKDIR /app

# Спочатку залежності (оптимізація кешу)
COPY package*.json ./

RUN npm ci --only=production

# Потім код
COPY . .

# Безпека: не запускати від root
USER node

# Метадані
EXPOSE 3000

CMD ["node", "app.js"]
```

Файл .dockerignore

При docker build Docker надсилає весь вміст директорії у Docker Engine. Якщо є 200 МБ node_modules - все буде надіслано, навіть якщо потім не скопійовано. Файл .dockerignore вказує, що не надсилати:

```
node_modules
.env
.git
.gitignore
*.md
.DS_Store
docker-compose*.yaml
```

Docker CLI: основні команди

Команди для роботи з образами

```
docker pull node:18-alpine # завантажити образ
docker images               # список образів
docker build -t my-app:1.0 . # зібрати образ
docker rmi my-app:1.0       # видалити образ
```

```
docker image prune -a # видалити невикористовувані
```

Команда **docker run** - детальний розбір

docker run - найважливіша команда. Розберемо всі опції:

```
docker run \  
  -d \                # detached: запустити у фоні  
  --name my-webapp \   # назва контейнера  
  -p 8080:3000 \       # port mapping: хост:контейнер  
  -e NODE_ENV=production \ # змінна оточення  
  -v my-volume:/app/data \ # підключити volume  
  --network my-network \ # мережа  
  --restart unless-stopped \ # автоперезапуск  
  my-app:1.0           # образ
```

Port mapping (-p 8080:3000): Браузер відкриває localhost:8080 → Docker перенаправляє у контейнер на порт 3000 → Node.js обробляє запит.

Управління контейнерами

```
docker ps                # запущені контейнери  
docker ps -a             # всі (включно зупинені)  
docker stop my-webapp    # зупинити (SIGTERM)  
docker start my-webapp    # запустити зупинений  
docker rm -f my-webapp    # видалити контейнер  
docker logs -f my-webapp  # логи в реальному часі  
docker exec -it my-webapp sh # увійти в оболонку  
docker stats             # моніторинг ресурсів
```

docker exec -it <container> sh - відкриває оболонку всередині запущеного контейнера, як SSH-підключення до сервера. Що можна перевірити: `ls -la /app/` (файли), `env | grep DB` (змінні оточення), `pg_isready -h database` (підключення до БД).

Docker Compose: оркестрація багатокомпонентних систем

Навіщо потрібен Docker Compose

У реальних проєктах додаток рідко складається з одного сервісу. Типова архітектура: Node.js API + PostgreSQL + Redis + Nginx. Для кожного потрібен окремий контейнер. Запуск через **docker run** - це чотири довгі команди. Docker Compose описує всю систему в одному файлі `docker-compose.yml` і дозволяє керувати нею кількома командами.

Структура `docker-compose.yml`

YAML (YAML Ain't Markup Language) - формат конфігурації з відступами пробілами (ніколи табуляцією). Стандарт - 2 пробіли на рівень.

```
version: "3.9"
```

```

services:
  web:
    # ім'я сервісу = hostname
    build: .
    # зібрати з Dockerfile
    ports:
      - "8080:3000"
    environment:
      - NODE_ENV=production
      - DB_HOST=database
    # 'database' = ім'я сервісу нижче
    depends_on:
      database:
        condition: service_healthy
    restart: unless-stopped
    networks:
      - backend

  database:
    image: postgres:15-alpine
    environment:
      - POSTGRES_PASSWORD=secret123
    volumes:
      - postgres-data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 10s
      timeout: 5s
      retries: 5
    networks:
      - backend

networks:
  backend:
    driver: bridge

volumes:
  postgres-data:
    # дані зберуться після down!

```

Основні команди Docker Compose

```

docker-compose up -d
# запустити у фоні
docker-compose up -d --build
# перезібрати образи і запустити
docker-compose down
# зупинити і видалити контейнери
docker-compose down -v
# + видалити volumes (дані БД!)
docker-compose ps
# стан сервісів

```

```

docker-compose logs -f          # логи в реальному часі
docker-compose logs -f web      # логи тільки одного сервісу
docker-compose exec web sh      # оболонка у запущеному сервісі
docker-compose restart web      # перезапустити один сервіс
docker-compose config           # перевірити yaml-файл

```

Docker Volumes: збереження даних

Чому контейнери «забувають» дані

При запуску контейнера Docker додає тонкий шар для запису (writable layer) поверх незмінних шарів Image. Всі зміни записуються лише у цей шар. При `docker rm` - writable layer видаляється разом із контейнером. Дані зникають!

Volume - це директорія на хост-машині, підключена до контейнера. Дані у Volume зберігаються незалежно від lifecycle контейнера.

Named Volume - для баз даних

Named Volume - Docker сам керує цим томом, зберігає у `/var/lib/docker/volumes/<name>/`. Рекомендується для баз даних і постійних даних.

```

# У docker-compose.yml
services:
  database:
    volumes:
      - postgres-data:/var/lib/postgresql/data

volumes:
  postgres-data:

# Команди:
docker volume ls          # список volumes
docker volume inspect postgres-data # деталі
docker volume rm postgres-data # видалити

```

Bind Mount - для розробки

Bind Mount - ви вказуєте конкретну директорію хосту, яка підключається до контейнера. Зміни на хості одразу видні у контейнері. Ідеально для hot-reload під час розробки: редагуєте файл у VS Code → зміни миттєво в контейнері → `nodemon` перезапускає сервер.

```

services:
  web:
    volumes:
      - ../app          # bind mount: проект -> /app

```

```
- /app/node_modules          # node_modules з образи!  
command: ["npm", "nodemon", "app.js"]
```

Трюк з /app/node_modules: це anonymous volume без шляху на хості. Потрібен, щоб node_modules з контейнера (Linux) не замінювались вашими node_modules з Windows/macOS - деякі бінарні модулі несумісні між ОС.

Мережі в Docker: як контейнери спілкуються

Вбудований DNS у Docker Compose

Проблема: веб-сервер повинен підключитись до бази даних. Яку адресу вказати? IP-адресу контейнера? Але при кожному docker-compose up контейнер може отримати інший IP!

Рішення: У Docker Compose кожен сервіс автоматично отримує DNS-ім'я, рівне імені сервісу. Якщо сервіс БД називається database - підключення: postgresql://database:5432/myapp. Docker сам визначає актуальну IP-адресу.

Bridge Network - стандартна мережа

Bridge Network (мостова мережа) - програмний комутатор у Docker. Ізолює контейнери від зовнішнього світу і від інших Docker-мереж, але дозволяє контейнерам у межах однієї мережі спілкуватись між собою.

Docker Compose автоматично створює одну bridge-мережу для всіх сервісів. Але хорошою практикою є явне оголошення мереж для ізоляції: frontend-мережа (Nginx + React), backend-мережа (API + DB). База даних не доступна з frontend напряму.

Типові налаштування мереж

```
networks:  
  frontend:          # доступна ззовні (через порти)  
    driver: bridge  
  backend:           # внутрішня, БД тут  
    driver: bridge  
    internal: true    # не доступна ззовні взагалі
```

Тип мережі	Опис	Коли використовувати	Особливість
bridge (за замовч.)	Програмний комутатор, DNS між сервісами	Більшість проєктів	Ізоляція від інших Compose-проєктів

host	Контейнер використовує мережу хосту напряму	Максимальна продуктивність (Linux only)	Немає ізоляції портів
none	Без мережі взагалі	Ізольовані обчислювальні задачі	Немає мережевого доступу

Електронні ресурси

1. Docker Official Documentation: docs.docker.com - повна документація з прикладами.
2. Play with Docker (PWD) - Docker у браузері без встановлення: labs.play-with-docker.com
3. Fireship - Docker in 100 Seconds (відео):youtu.be/Gjnup-PuquQ
4. TechWorld with Nana - Docker Tutorial for Beginners (3 год, відео):youtu.be/3c-iBn73dDE

Підсумок лекції

#	Ключова думка	Практичний висновок
1	Docker вирішує проблему «Works on my machine» — контейнер однаково запускається на Windows, macOS і Linux-сервері.	Пишіть Dockerfile для кожного проєкту з першого дня, не покладайтесь на ручне налаштування.
2	Dockerfile — це рецепт; Image — шаблон; Container — запущений екземпляр. Ці три поняття нерозривно пов'язані.	Запам'ятайте ланцюжок: код → Dockerfile → docker build → Image → docker run → Container.
3	Шари Image кешуються: залежності (package.json + npm install) копіюйте перед кодом, щоб прискорити збірку.	Не змінюйте порядок COPY/RUN у Dockerfile без потреби — це руйнує кеш і уповільнює CI/CD.
4	Volume — єдиний спосіб зберегти дані між перезапусками. Без нього база даних очищується при кожному docker-compose down.	Завжди оголошуйте Named Volume для PostgreSQL/MySQL у docker-compose.yml.
5	Docker Compose описує всю багатокомпонентну систему (API + БД + Redis + Nginx) в одному YAML-файлі та запускає її однією командою.	Використовуйте depends_on з condition: service_healthy, щоб API не падав до готовності бази.
6	Мережі ізолюють сервіси: backend-мережа (API + БД) недоступна з frontend. DNS за іменем сервісу замінює IP-адреси.	Оголошуйте окремі networks у Compose-файлі явно для кращої безпеки.

#	Ключова думка	Практичний висновок
7	Секрети (паролі, API-ключі) — лише у файлі <code>.env</code> , який внесений до <code>.gitignore</code> та <code>.dockerignore</code> . Ніколи у <code>Dockerfile</code> або коді.	Перевіряйте <code>.dockerignore</code> перед кожним першим <code>push</code> — один витік може коштувати десятки тисяч доларів.
8	Docker — не конкурент VM, а доповнення. Типова архітектура: хмарна VM → Docker Engine → кілька контейнерів із мікросервісами.	Контейнери підходять для мікросервісів і CI/CD; VM — для повної ізоляції ОС.

КОНТРОЛЬНІ ПИТАННЯ

Блок 1. Основні поняття Docker

1. Що таке Docker і яку проблему він вирішує? Поясніть концепцію «Works on my machine» та наведіть три типові причини її виникнення.
2. Поясніть різницю між поняттями `Dockerfile`, `Image` та `Container`. Яка аналогія з кулінарією або ООП допомагає зрозуміти їхній взаємозв'язок?
3. Що таке `Docker Volume` і навіщо він потрібен? Що відбувається з даними бази даних, якщо не використовувати `Volume` при `docker-compose down`?
4. Що таке `Docker Network` і яку проблему вирішує вбудований `DNS` у `Docker Compose`? Чому не можна просто вказати IP-адресу контейнера при підключенні?
5. Що таке `Docker Compose` і яку перевагу він дає порівняно з ручним запуском кількох `docker run` команд?

Блок 2. Порівняння VM та Docker

6. Порівняйте VM та Docker-контейнер за часом запуску, розміром образу, споживанням RAM та рівнем ізоляції. У яких випадках VM доцільніша за контейнери?
7. Які два механізми ядра Linux забезпечують ізоляцію Docker-контейнерів? Що кожен з них робить?
8. Чи є Docker і VM конкурентами? Опишіть типову хмарну архітектуру, де вони використовуються разом.

Блок 3. Dockerfile

9. Назвіть і поясніть призначення не менше семи інструкцій `Dockerfile` (`FROM`, `WORKDIR`, `COPY`, `RUN`, `ENV`, `EXPOSE`, `CMD`, `USER`).
10. Чому у `Dockerfile` рекомендується спочатку копіювати `package.json` і виконувати `npm install`, і лише потім копіювати весь код? Що таке кешування шарів?
11. Навіщо у `Dockerfile` використовується інструкція `USER`? Яка загроза безпеки виникає, якщо запускати контейнер від `root`?

12. Що таке файл `.dockerignore` і які категорії файлів обов'язково потрібно до нього включати? Чому важливо виключати `node_modules`?

Блок 4. Docker CLI

13. Розберіть команду `docker run -d --name my-app -p 8080:3000 -e NODE_ENV=production -v my-vol:/app/data my-app:1.0`. Поясніть значення кожного прапорця.
14. Що робить команда `docker exec -it <container> sh`? Для чого вона використовується на практиці?
15. Яка різниця між командами `docker stop` і `docker rm`? Коли використовується кожна?

Блок 5. Docker Compose

16. Поясніть структуру файлу `docker-compose.yml`: що описується в розділах `services`, `networks` та `volumes`?
17. Чим відрізняються команди `docker-compose down` та `docker-compose down -v`? Коли використання другої є небезпечним?
18. Що означає параметр `condition: service_healthy` у `depends_on`? Яку проблему він вирішує і що потрібно додатково налаштувати у сервісі бази даних?
19. Як Docker Compose забезпечує зв'язок між сервісами без вказання IP-адрес? Наведіть приклад рядка підключення до бази даних.

Блок 6. Volumes та мережі

20. Порівняйте Named Volume та Bind Mount: у чому різниця, де зберігаються дані та в яких ситуаціях використовується кожен тип?
21. Навіщо при використанні Bind Mount у розробці додають рядок `- /app/node_modules` без шляху на хості? Яку проблему це вирішує?
22. Поясніть різницю між типами мереж `bridge`, `host` та `none`. Коли доцільно використовувати `internal: true` для мережі?

Блок 7. Типові помилки та безпека

23. Опишіть шість типових помилок при роботі з Docker та їхні наслідки (за зведеною таблицею лекції). Яка з них може призвести до найбільших фінансових втрат?
24. Чому паролі та API-ключі не можна зберігати у Dockerfile в інструкції `ENV`? Як правильно передавати секрети у контейнер?
25. Поясніть реальний інцидент 2022 року, описаний у лекції. Яких правил потрібно дотримуватись, щоб уникнути подібного?

Питання підвищеної складності

26. Намалюйте або опишіть словами архітектуру типової веб-системи (Node.js API + PostgreSQL + Nginx), розгорнутої через Docker Compose: які сервіси існують, які мережі їх з'єднують, де підключені Volume?

27. Чому у CI/CD пайплайнах (GitHub Actions, GitLab CI) Docker є обов'язковою технологією? Як кешування шарів Image впливає на швидкість автоматичного тестування і деплою?

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Проектування інформаційних систем: методичні вказівки /Уклад. О.В.Ізмайлова , – Київ : КНУБА , 2023 . – 28 с .
<https://repository.knuba.edu.ua/items/f1ee0cf1-e830-43a7-b33e-dc2af6f40378>
2. Ніколюк П. К. Моделювання систем : навч. посіб. для здобувачів вищої освіти спеціальності 122 Комп'ютерні науки. Вінниця: ДонНУ, 2023. 228 с.
3. Міца О.В., Лавер В.О. Системний аналіз : навч.-метод. посіб. / О.В. Міца, В.О. Лавер. Ужгород : вид-во ПП «АУТДОР - ШАРК», 2021. 63 с.
4. Ізмайлова, О. В. Проектування інформаційних систем: навч. посібник : для студ. галузі знань 12 "Інформаційні технології" - Київ : КНУБА, 2022. - 87 с.
5. Литвин В. В., Пасічник В. В., Шаховська Н. Б. Проектування інформаційних систем: Навчальний посібник. Львів: «Магнолія-2006», 2021. 380 с.
6. Новіков Ф. В. Техніко-економічне обґрунтування сучасних технологій виробництва : навч. посібник / Ф. В. Новіков, Д. Ф. Новіков, О. А. Єрмоленко та ін. – Дніпро: ЛПРА, 2022. – 256 с. Режим доступу: <http://www.repository.hneu.edu.ua/handle/123456789/28117>
7. Сучасні інформаційні технології та системи [Електронний ресурс] : монографія / Н. Г. Аксак, Л. Е. Гризун, О. В. Щербаков [та ін.] ; за заг.ред. Пономаренка В. С. – Харків : ХНЕУ ім. С. Кузнеця, 2022. – 270 с. Режим доступу: <http://repository.hneu.edu.ua/handle/123456789/29233>
8. Теорія систем та системний аналіз : курс лекцій / В. В. Тютюник, О. О. Писклакова. – Харків : Друкарня Мадрид, 2020. – 108 с.
9. Рубан І. В. Особливості створення системи підтримки прийняття антикризових рішень в умовах невизначеності вхідної інформації при надзвичайних ситуаціях / І. В. Рубан, В. В. Тютюник, О. О. Тютюник // Сучасні інформаційні технології у сфері безпеки та оборони. – 2021. – №1(40). – С. 75-84. Режим доступу: <http://repository.hneu.edu.ua/handle/123456789/25600>
10. Вихлюк Я. І. Моделювання складних систем : посібник / Я. І. Вихлюк, Р.М. Камінський, В. В. Пасічник. – Львів : Вид-во «Новий Світ –2000», 2020. – 404 с., 2023. – 39 с.
11. Зелінська О. В. Інформаційні системи та технології в галузі : навч. посіб. / О. В. Зелінська, Н. А. Потапова, Л. О. Волонтир. – Вінниця : ВНАУ, 2020. – 253 с

12. Крєневич А. П. Алгоритми і структури даних : підручник / А. П. Крєневич. – Київ : ВПЦ «Київський Університет», 2021. – 200 с
13. Методологія інформаційних систем та баз даних: теоретичний і практичний підходи : навч. посіб. / уклад. Ю. О. Ушенко, М. Л. Ковальчук, М. С. Гавриляк, А. Л. Нєгрич. – Чернівці : Чернівецький нац. ун-т ім. Ю. Федьковича, 2021. – 240 с.
14. Мінухін С. Метод збільшення продуктивності Apache Spark на основі сегментування даних і налаштувань конфігураційних параметрів [Електронний ресурс] / С. Мінухін, Н. Коптілов // Сучасний стан наукових досліджень та технологій промисловості. – 2024. – № 1(27). – С. 128–139. – Режим доступу : <https://doi.org/10.30837/ITSSI.2024.27.128>
15. Kief Morris. Infrastructure as Code. O'Reilly Media, 2020 – 462 p. Michael Kreef, Mitchell Hashimoto. Terraform Cookbook: Efficiently Define, Launch, and Manage Infrastructure as Code Across Various Cloud Platforms. Packt Publishing, 2020 – 366 p.
16. Thomas Kurian Tekanat. Datadog Cloud Monitoring Quick Start Guide. Packt Publishing, 2021 – 318 p.
17. Brendan Burns, Joe Beda, Kelsey Hightower, Lachlan Evenson. Kubernetes: Up and Running, 3rd Edition. O'Reilly Media, 2022 – 540 p.
18. Raphael Hertzog, Jim O’Gorman, Mati Aharoni, Joe O’Gorman. Kali Linux Revealed: Mastering the Penetration Testing Distribution. OffSec Press, 2021 – 346 p.

Інтернет-ресурси:

1. Академія web розробки MDN [Електронний ресурс]. – Режим доступу: <https://developer.mozilla.org/>
2. HTML Tutorial [Електронний ресурс]. – Режим доступу: <https://www.tutorialspoint.com/html/index.htm>
3. CSS Tutorial [Електронний ресурс]. – Режим доступу: <https://www.tutorialspoint.com/css/index.htm>
4. Javascript Tutorial [Електронний ресурс]. – Режим доступу: <https://www.tutorialspoint.com/javascript/index.htm>
5. SQL Tutorial [Електронний ресурс]. – Режим доступу: <https://www.tutorialspoint.com/sql/index.htm>
6. Навчальна платформа Coursera, Secure Software Design [Електронний ресурс]. – Режим доступу до ресурсу: <https://www.coursera.org/programs/program->

natsional-nii-tiekhnichnii-univiersitiet-ukrayini-kiyivs-kii/specializations/secure-software-design

7. Навчальна платформа Coursera, Information Systems [Електронний ресурс]. – Режим доступу до ресурсу: <https://www.coursera.org/programs/program-natsional-nii-tiekhnichnii-univiersitiet-ukrayini-kiyivs-kii/specializations/information-systems>

8. Навчальна платформа Udemy, Software Architecture & System Design Practical Case Studies [Електронний ресурс]. – Режим доступу до ресурсу: <https://ua.udemy.com/course/software-architecture-system-design-practical-case-studies>

9. Навчальна платформа Udemy, Software Architecture & Design of Modern Large Scale Systems [Електронний ресурс]. – Режим доступу до ресурсу: <https://ua.udemy.com/course/software-architecture-design-of-modern-large-scale-systems/>

10. Навчальна платформа Udemy, Software Architecture & Technology of Large-Scale Systems [Електронний ресурс]. – Режим доступу до ресурсу: <https://ua.udemy.com/course/developer-to-architect/>

11. Навчальна платформа Wintellectnow, Architecting Distributed Cloud Applications [Електронний ресурс]. – Режим доступу до ресурсу: <https://www.wintellectnow.com/Home/SeriesDetail?seriesId=architecting-distributed-cloud-applications>

Інформаційні ресурси

1. Електронна бібліотека ЛьвДУВС: URL:<https://www1.lvduvs.edu.ua/elektronnyi-kataloh>

2. Державна науково-технічна бібліотека України. URL: <http://www.gntb.nt.org>.

Сайт національної бібліотеки України імені В.І.Вернадського. URL: <http://www.nbuv.gov.ua>.

3. Бібліотека ПДАУ. URL: <https://www.pdaa.edu.ua/content/biblioteka>.

4. Atlassian solutions. URL: <https://www.atlassian.com/>

5. Bitbucket - The code collaboration platform for modern software teams.

URL:<https://bitbucket.org/>

6. Docker. URL: <https://www.docker.com/>

7. drawSQL - Design, visualize and collaborate on entity relationship diagrams for your databases. URL:

<https://drawsql.app/>

8. git – fast version control. URL: <https://git-scm.com/>

9. GitHub - Where the world builds software. URL: <https://github.com/>

10. GitLab – the DevOps Platform. URL: <https://about.gitlab.com/>

11. Kubernetes. URL: <https://kubernetes.io/>

12. Trello. URL: [ps://trello.com/home](https://trello.com/home)

13. Visual Studio Code. URL: <https://code.visualstudio.com/>